1978

MICRO-COBOL : a subset of Navy standard HYPO-COBOL for micro-computers.

Mylet, Philip Russell

MICRO-COBOL
A SUBSET OF
NAVY STANDARD HYPO-COBOL
FOR MICRO-COMPUTERS

Philip Russell Mylet

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

MICRO-COBOL
A SUBSET OF
NAVY STANDARD HYPO-COBOL
FOR MICRO-COMPUTERS

by

Philip Russell Mylet

September 1978

Thesis Advisor: G. A. Kildall

Approved for public release; distribution unlimited.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* MICRO-COBOL a Subset of Navy Standard Hypo-Cobol for Micro-Computers		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; September 1978
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Philip Russell Mylet		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE September 1978
		13. NUMBER OF PAGES 169
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

MICRO-COBOL
Navy Standard Hypo-Cobol
Micro-Computers
Compiler

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

A MICRO-COBOL interpretive compiler has been implemented on an 8080 micro-computer based system running under CP/M. The implementation is a subset of ADPESO standard HYPO-COBOL in that the interprogram communication module has not been included. HYPO-COBOL provides nucleus level constructs and file options from the ANSII COBOL package along with the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

PERFORM UNTIL construct from a higher level to give increased structural control. MICRO-COBOL can be executed on an 8080 or Z-80 micro-computer system with 16K of memory. Although largely completed and tested, all features are not implemented. File I/O features have not been tested and the numeric edit instruction has not been implemented in the interpreter.

MICRO-COBOL
A Subset of
Navy Standard HYPO-COBOL
for Micro-Computers

by

Philip Russell Mylet
B.S., Pennsylvania State University, 1967

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1978

[illegible]
[illegible] 8
[illegible]

ABSTRACT

A MICRO-COBOL interpretive compiler has been implemented on an 8080 micro-computer based system running under CP/M. The implementation is a subset of ADPESO standard HYPO-COBOL in that the interprogram communication module has not been included. HYPO-COBOL provides nucleus level constructs and file options from the ANSII COBOL package along with the PERFORM UNTIL construct from a higher level to give increased structural control. MICRO-COBOL can be executed on an 8080 or Z-80 micro-computer system with 16K of memory. Although largely completed and tested, all features are not implemented. File I/O features have not been tested and the numeric edit instruction has not been implemented in the interpreter.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I wish to express my appreciation to my advisor, Gary Kildall who cheerfully accepted the responsibilities of thesis advisor while on leave of absence. My thanks also to John Pierce of Digital Research for his contributions and hours of assistance early in the project. Finally, I wish to express my gratitude to Mark Moranville who continuously provided technical assistance and moral support during the times when it was most needed.

I. INTRODUCTION

A. BACKGROUND

MICRO-COBOL is an implementation of ADPESO standard MYPO-COBOL with the major exception that the interprogram communication module is not included. It has been implemented as an interpretive compiler in that the compiler itself generates intermediate code which is then executed by a separate interpreter program. Both compiler and interpreter run under CP/M on an 8080 or Z-80 micro-computer system with 16K of memory. Much credit for this work goes to Allen S. Craig who did the original design and implementation of MICRO-COBOL for his thesis submitted in March 1977. Craig's work is contained in Reference 1. Most of the coding had been completed, but many of the constructs did not work or worked incorrectly. Since much of the compiler had not been debugged and some areas not completed, thesis work was continued in March 1978 with the goal of producing a working MICRO-COBOL compiler and interpreter.

B. APPROACH

As a first step, the program listings and thesis were studied to gain familiarity with the original project goals and resolve several areas of conflict between the thesis and the listings. The remaining effort consisted of running test programs, isolating bugs, and making additions, corrections

and small design changes. The problems discovered were primarily errors in the code, however, there were also missing routines and grammar problems which necessitated reconstructing the original grammar. Appendix D lists the features that did not work at the start of this project and the bugs that are known to remain.

The HYPO-COBOL Compiler Validation System (HCCVS) was obtained from the Automatic Data Processsing Equipment Selection Office (ADPESO) to be used in testing the compiler. The HCCVS is intended to determine the degree to which the individual language elements conform to the HYPO-COBOL Specification. The validation system is made up of audit routines, their related data, and an executive routine which prepares the audit routines for compilation. Each audit routine is a HYPO-COBOL program which includes tests and supporting procedures that print out the results of each test. The audit routines collectively test the features of the HYPO-COBOL Language Specification. Since MICRO-COBOL does not support the interprogram communication module feature of HYPO-COBOL, the HCCVS is not useful in its existing form; however, it contains numerous routines which can be used to create small test programs that should run on MICRO-COBOL as it currently exists.

A language construct in question was tested by writing a test program, compiling it, and executing it on the interpreter. If problems were encountered, the intermediate code

was examined to determine if the difficulty was in the compiler or the interpreter. Having made this determination, the program was examined to isolate the bad code using SID (see Reference 12). Changes were then made and the source program recompiled using the ISIS editor and the PLM80 compiler on the INTEL MDS System. Appendix B describes the procedure used to construct the executable compiler and interpreter files from the edited PLM80 source files.

The following sections describe the implementation of the compiler and interpreter. This material should be read in conjunction with Reference 1 which contains additional background information.

II. MICRO-COBOL INTERPRETER

A. GENERAL DESCRIPTION

The following sections describe the MICRO-COBOL pseudo-machine architecture in terms of allocated memory areas and pseudo-machine operations. The machine operators contain all of the information required to perform one complete action required by the language. The machine contains multiple parameter operators and a program counter that addresses the next instruction to be executed. Three eighteen digit registers are used for arithmetic and logic operations. A subscript stack is used to compute subscript locations, and a set of flags are used to pass branching information from one instruction to another. The registers allow manipulation of signed numbers of up to eighteen decimal digits in length. Included in their representation is a sign indicator and the position of the assumed decimal point for the currently loaded number. The HYPO-COBOL specification requires that there be no loss of precision for operations on numbers having eighteen significant digits. Numbers are represented in "DISPLAY" and "packed decimal" formats. DISPLAY format numbers are represented in memory in ASCII and may have separate signs indicated by "+" and "-" or may have a "zone" indicator, denoting a negative sign. In packed decimal format the numbers are represented in memory as sequential digit pairs and the sign is indicated in the right-most position.

B. MEMORY ORGANIZATION

Memory is divided into three major sections: (1) the data areas defined by the DATA DIVISION statements, (2) the code area, (3) and the constants area. No particular order of these sections is required. The first two areas assume the ability to both read and write, but the third only requires the ability to be read. The code area requires write capability because several instructions store branch addresses and return addresses during execution.

The data area contains variables defined by the DATA DIVISION statements, constants set in the WORKING STORAGE SECTION, and all file control blocks and buffers. These elements will be manipulated by the machine as each instruction is executed.

C. INTERPRETER INSTRUCTIONS

1. Format

All of the interpreter instructions consist of an instruction number followed by a list of parameters. The following sections describe the instructions, list the required parameters, and describe the actions taken by the machine in executing each instruction. In each case, parameters are denoted informally by the parameter name enclosed in brackets. The BRN branching instruction, for example, uses the single parameter <branch address> which is the target of the unconditional branch.

As each instruction number is fetched from memory, the program counter is incremented by one. The program counter is then either incremented to the next instruction number, or a branch is taken.

The three eighteen digit registers which are used by the instructions covered in the following section are referred to as registers zero, one, and two.

2. Arithmetic Operations

There are five arithmetic instructions which act upon the three registers. In all cases, the result is placed in register two. Operations are allowed to destroy the input values during the process of creating a result, therefore, a number loaded into a register is not available for a subsequent operation.

ADD: (addition). Sum the contents of register zero and register one.

Parameters: no parameters are required.

SUB: (subtract). Subtract register zero from register one.

Parameters: no parameters are required.

MUL: (multiply). Multiply register zero by register one.

Parameters: no parameters required.

DIV: (divide). Divide register one by the value in register zero. The remainder is not retained.

Parameters: no parameters are required.

RND: (round). Round register two to the last significant significant decimal place.

Parameters: no parameters are required.

3. Branching

The machine contains the following flags which are used by the conditional instructions covered in this section.

BRANCH flag -- indicates if a branch is to be taken;

END OF RECORD flag -- indicates that an end of input condition has been reached when an attempt was made to read input;

OVERFLOW flag -- indicates the loss of information from a register due to a number exceeding the available size;

INVALID flag -- indicates an invalid action in writing to a direct access storage device.

All of the branch instructions are executed by changing the value of the program counter. Some are unconditional branches and some test for condition flags which are set by other instructions. A conditional branch is executed by testing the branch flag which is initialized to false. A true value causes a branch by changing the program counter to the value of the branch address. The branch flag is then reset to false. A false value causes the program counter to be incremented to the next sequential instruction.

BRN: (branch to an address). Load the program counter with the <branch address>.

Parameters: <branch address>

The next three incstructions share a common format. The memory field addressed by the <memory address> is checked for the <address length>, and if all the characters match the test condition, the branch flag is complemented.

Parameters: <memory address> <address length> <branch address>

CAL: (compare alphabetic). Compare a memory field for alphabetic characters.

CNS: (compare numeric signed). Compare a field for numeric characters allowing for a sign character.

CNU: (compare numeric unsigned). Compare a field for numeric characters only.

DEC: (decrement a counter and branch if zero). Decrement the value of the <address counter> by one; if the result is zero before or after the decrement, the program counter is set to the <branch address>. If the result is not zero, the program counter is incremented by four.

Parameters: <address counter> <branch address>

EOR: (branch on end-of-records flag). If the end-of-records flag is true, it is set to false and the program counter is set to the <branch address>. If false, the program counter is incremented by two.

Parameters: <branch address>

GDP: (go to - depending on). The memory location addressed by the <number address> is read for the number of bytes indicated by the <memory length> . This number indicates which of the <branch addresses> is to be used.

The first parameter is a bound on the number of branch addresses. If the number is within the range, the program counter is set to the indicated address. An out-of-bounds value causes the program counter to be advanced to the next sequential instruction.

Parameters: <bound number - byte> <memory length> <memory address> <branch addr-1> <branch addr-2> ... <branch addr-n>

INV: (branch if invalid-file-action flag true). If the invalid-file-action flag is true, then it is set to false, and the program counter is set to the branch address. If it is false, the program counter is incremented by two.

Parameters: <branch address>

PER: (perform). The code address addressed by the <change address> is loaded with the value of the <return address>. The program counter is then set to the <branch address>.

Parameters: <branch address> <change address> <return address>

RET: (return). If the value of the <branch address> is not zero, then the program counter is set to its value, and the <branch address> is set to zero. If the <branch address> is zero, the program counter is incremented by two.

Parameters: <branch address>

REQ: (register equal). This instruction checks for a zero value in register two. If it is zero, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

RGT: (register greater than). Register two is checked for a negative sign. If present, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

SER: (branch on size error). If the overflow flag is true, then the program counter is set to the branch address, and the overflow flag is set to false. If it is false, then the program counter is incremented by two.

Parameters: <branch address>.

The next three instructions are of similar form in that they compare two strings and set the branch flag if the condition is true.

Parameters: <string addr-1> <string addr-2> <length - address> <branch address>

SEQ: (strings equal). The condition is true if the strings are equal.

SGT: (string greater than). The condition is true if string one is greater than string two.

SLT: (string less than). The condition is true if string one is less than string two.

4. Moves

The machine supports a variety of move operations for various formats and types of data. It does not support direct moves of numeric data from one memory field to another. Instead, all of the numeric moves go through the registers.

The next seven instructions all perform the same function. They load a register with a numeric value and

differ only in the type of number that they expect to see in memory at the <number address>. All seven instructions cause the program counter to be incremented by five. Their common format is given below.

Parameters: <number address> <byte length> <byte decimal count> <byte register to load>

LOD: (load literal). Register two is loaded with a constant value. The decimal point indicator is not set in this instruction. The literal will have an actual decimal point in the string if required.

LD1: (load numeric). Load a numeric field.

LD2: (load postfix numeric). Load a numeric field with an internal trailing sign.

LD3: (load prefix numeric). Load a numeric field with an internal leading sign.

LD4: (load separated postfix numeric). Load a numeric field with a separate leading sign.

LD5: (load separated prefix numeric). Load a numeric field with a separate trailing sign.

LD6: (load packed numeric). Load a packed numeric field.

MED: (move into alphanumeric edited field). The edit mask is loaded into the <to address> to set up the move, and then the <from address> information is loaded. The program counter is incremented by ten.

Parameters: <to address> <from address> <length of move> <edit mask address> <edit mask length>

MNE: (move into a numeric edited field). First the edit mask is loaded into the receiving field, and then the information is loaded. Any decimal point alignment required will be performed. If truncation of significant digits is a side effect, the overflow flag is not set. The program counter is incremented by twelve.
Parameters: <to address> <from address> <address length of move> <edit mask address> <address mask length> <byte to decimal count> <byte from decimal count>

MOV: (move into an alphanumeric field). The memory field given by the <to address> is filled by the from field for the <move length> and then filled with blanks in the following positions for the <fill count>.
Parameters: <to address> <from address> <address move length> <address fill count>

STI: (store immediate register two). The contents of register two are stored into register zero and the decimal count and sign are indicators set.
Parameters: none.

The store instructions are grouped in the same order as the load instructions. Register two is stored into memory at the indicated location. Alignment is performed and any truncation of leading digits causes the overflow flag to be set. All five of the store instructions cause the program counter to be incremented by four. The format for these instructions is as follows.
Parameters: <address to store into> <byte length> <byte decimal count>

MNE: (move into a numeric edited field). First the edit mask is loaded into the receiving field, and then the information is loaded. Any decimal point alignment required will be performed. If truncation of significant digits is a side effect, the overflow flag is not set. The program counter is incremented by twelve.

Parameters: <to address> <from address> <address length of move> <edit mask address> <address mask length> <byte to decimal count> <byte from decimal count>

MOV: (move into an alphanumeric field). The memory field given by the <to address> is filled by the from field for the <move length> and then filled with blanks in the following positions for the <fill count>.

STI: (store immediate register two). The contents of register two are stored into register zero and the decimal count and sign are indicators set.

Parameters: none.

The store instructions are grouped in the same order as the load instructions. Register two is stored into memory at the indicated location. Alignment is performed and any truncation of leading digits causes the overflow flag to be set. All five of the store instructions cause the program counter to be incremented by four. The format for these instructions is as follows.

Parameters: <address to store into> <byte length> <byte decimal count>

ST0: (store numeric). Store into a numeric field.

ST1: (store postfix numeric). Store into a numeric field with an internal trailing sign.

ST2: (store prefix numeric). Store into a numeric field with an internal leading sign.

ST3: (store separated postfix numeric). Store into a numeric field with a separate trailing sign.

ST4: (store separated prefix numeric). Store into a numeric field with a separate leading sign.

ST5: (store packed numeric). Store into a packed numeric field.

5. Input-Output

The following instructions perform input and output operations. Files are defined as having the following characteristics: they are either sequential or random and, in general, files created in one mode are not required to be readable in the other mode. Standard files consist of fixed length records, and variable length files need not be readable in a random mode. Further, there must be some character or character string that delimits a variable length record.

ACC: (accept). Read from the system input device into memory at the location given by the memory address . The program counter is incremented by three.

Parameters: <memory address> <byte length of read>

CLS: (close). Close the file whose file control block is addressed by the <fcb address>. The program counter is incremented by two.

Parameters: <fcb address>

DIS: (display). Print the contents of the data field pointed to by <memory address> on the system output device for the indicated length. The program counter is incremented by three.

Parameters: <memory address> <byte length>

There are three open instructions with the same format. In each case, the file defined by the file control block referenced will be opened by the mode indicated. The program counter is incremented by two.

OPN: (open a file for input).

OP1: (open a file for output).

OP2: (open a file for both input and output). This is only valid for files on a random access device.

The following file actions all share the same format. Each performs a file action on the file referenced by the file control block. The record to be acted upon is given by the record address . The program counter is incremented by six.

Parameters: <fcb address> <record address> <record length - address>

DLS: (delete a record from a sequential file). Remove the record that was just read from the file. The file is required to be open in the input-output mode.

RDF: (read a sequential file). Read the next record into the memory area.

WTF: (write a record to a sequential file). Append a new record to the file.

RVL: (read a variable length record).

WVL: (write a variable length record).

RWS: (rewrite sequential). The rewrite operation writes a record from memory to the file, overlaying the last record that was read from the device. The file must be open in the input-output mode.

The following file actions require random files rather than sequential files. They all make use of a random file pointer which consists of a <relative address> and a <relative length>. The memory field holds the number to be used in disk operations or contains the relative record number of the last disk action. The relative record number is an index into the file which addresses the record being accessed. After the file action, the program counter is incremented by nine.

Parameters: <fcb address> <record address> <record length - address> <relative address> <relative length - byte>.

DLR: (delete a random record). Delete the record addressed by the relative record number.

RRR: (read random relative). Read a random record relative to the record number.

RRS: (read random sequential). Read the next sequential record from a random file. The relative record number of the record read is loaded into the memory reference.

RWR: (rewrite a random record).

WRR: (write random relative). Write a record into the area indicated by the memory reference.

WRS: (write random sequential). Write the next sequential record to a random file. The relative record number is returned.

6. <u>Special Instructions</u>

The remaining instructions perform special functions required by the machine that do not relate to any of the previous groups.

NEG: (negate). Complement the value of the branch flag.

Parameters: no parameters are required.

LDI: (load a code address direct). Load the code address located five bytes after the LDI instruction with the contents of <memory address> after it has been converted to hexidecimal.

Parameters: <memory address> <length - byte>

SCR: (calculate a subscript). Load the subscript stack with the value indicated from memory. The address loaded into the stack is the <initial address> plus an offset. Multiplying the <field length> by the number in the <memory reference> gives the offset value.

Parameters: <initial address> <field length> <memory reference> <memory length> <stack level>

STD: (stop display). Display the indicated information and then terminate the actions of the machine.
Parameters: <memory address> <length - byte>

STP: (stop). Terminate the actions of the machine.
Parameters: no parameters are required.

The following instructions are used in setting up the machine environment and cannot be used in the normal execution of the machine.

BST: (backstuff). Resolve a reference to a label. Labels may be referenced prior to their definition, requiring a chain of resolution addresses to be maintained in the code. The latest location to be resolved is maintained in the symbol table and a pointer at that location indicates the next previous location to be resolved. A zero pointer indicates no prior occurrences of the label. The code address referenced by <change address> is examined and if it contains zero, it is loaded with the new address . If it is not zero, then the contents are saved, and the process is repeated with the saved value as the change address after loading the <new address>.
Parameters: <change address> <new address>

INT: (initialize memory). Load memory with the <input string> for the given length at the <memory address>.
Parameters: <memory address> <address length> <input string>

SCD: (start code). Set the initial value of the program counter.
Parameters: <start address>

TER: (terminate). Terminate the initialization process and start executing code.

Parameters: no parameters are required.

III. MICRO-COBOL COMPILER

A. GENERAL

The compiler is designed to read the source language statements from a diskette, extract the needed information for the symbol table, and write the output code back onto the diskette all in one pass. The compiler is defined in two parts which run in succession. Part one builds the symbol table and leaves it in memory to be used by part two. The output from part two of the compiler is the intermediate code file.

B. CONTROL FLOW

After part one of the compiler has completed its task it loads part two without operator intervention. Internal control of the compiler is the same for both part one and two. The parser is called after initialization and runs until it either finishes its task or reaches an unrecoverable error state. The major subroutines in the compiler are the scanner and the production case statement which are both controlled by the parser.

C. INTERNAL STRUCTURES

The major internal structure is the symbol table, which was designed as a list where the elements in the list are the descriptions of the various symbols in the program. As

new symbols are encountered they are added to the end of the list. Symbols already in the list can be accessed through the use of a "current symbol pointer". The location of items in the list is determined by checking the identifier against a hash table that points to the first entry in the symbol table with that hash code. A chain of collision addresses is maintained in the symbol table which links entries which have the same hash value. All of the items in the symbol table contain the following information: a collision field, a type field, the length of the identifier, and the address of the item. If an item in the symbol table is a data field, the following information is included in the table: the length of the item, the level of the data field, an optional decimal count, an optional multiple occurrence count, and the address of the edit field, if required. If the item is a file name then the following additional information is included: the file record length, the file control block address, and the optional symbol table location of the relative record pointer. If the item is a label, then the only additional information is the location of the return instruction at the end of the paragraph or section.

In addition to the symbol table, two stacks are used for storing information: the level stack and the identifier stack. In both cases, they are used to hold pointers to entries in the symbol table. The identifier stack keeps track of multiple identifier occurrences in such statements

as the GO TO DEPENDING statement. The level stack is used to hold information about the levels that make up a record description.

The parser has control of a set of stacks that are used in the manipulation of the parse states. In addition to the state stack that is required by the parser, part one has a value stack while part two has two different value stacks that operate in parallel with the parser state stack. The use of these stacks is described below.

D. PART ONE

The first part of the compiler is primarily concerned with building the symbol table that will be used by the second part. The actions corresponding to each parse step are explained in the sections that follow. In each case, the grammar rule that is being applied is given, and an explanation of what program actions take place for that step has been included. In describing the actions taken for each parse step there has been no attempt to describe how the symbol table is constructed or how the values are preserved on the stack. The intent of this section is to describe what information needs to be retained and at what point in the parse it can be determined. Where no action is required for a given statement, or where the only action is to save the contents of the top of the stack, no explanation is given. Questions regarding the actual manipulation of information should be resolved by consulting the programs.

```
 1  <program> ::= <id-div> <e-div> <d-div> PROCEDURE
      Reading the word PROCEDURE terminates the first part
      of the compiler.
 2  <id-div> ::= IDENTIFICATION DIVISION.  PROGRAM-ID.
                 <comment> . <auth> <date> <sec>
 2  <auth> ::= AUTHOR . <comment> .
              | <empty>
 4
 5  <date> ::= DATE-WRITTEN . <comment> .
 6            | <empty>
 7  <sec> ::= SECURITY . <comment> .
 8           | <empty>
 9  <comment> ::= <input>
10               | <comment> <input>
11   <e-div> ::= ENVIRONMENT DIVISION . CONFIGURATION SECTION.
                 <scr-obj> <i-o>
12  <src-obj> ::= SOURCE-COMPUTER . <comment> <debug> .
                  OBJECT-COMPUTER . <comment> .
13  <debug> ::= DEBUGGING MODE
      Set a scanner toggle so that debug lines will be
      read.
14             | <empty>
15  <i-o> ::= INPUT-OUTPUT SECTION . FILE-CONTROL .
             <file-control-list> <id<
16           | <empty>
17  <file-control-list> ::= <file-control-entry>
18                        | <file-control-list> <file-
                            control-entry>
```

19 <file-control-entry> ::= SELECT <id> <attribute-list> .

At this point all of the information about the file has been collected and the type of the file can be determined. File attributes are checked for compatability and entered in the symbol table.

20 <attribute-list> ::= <one attrib>

21 | <attribute-list> <one attrib>

22 <one-attrib> ::= ORGANIZATION <org-type>

23 | ACCESS <acc-type> <relative>

24 | ASSIGN <input>

A file control block is built for the file using an INT operator.

25 <org-type> ::= SEQUENTIAL

No information needs to be stored since the default file organization is sequential.

26 | RELATIVE

The relative attribute is saved for production 19.

27 <acc-type> ::= SEQUENTIAL

This is the default.

28 | RANDOM

The random access mode needs to be saved for production 19.

29 <relative> ::= RELATIVE <id>

The pointer to the identifier will be retained by the current symbol pointer, so this production only saves a flag on the stack indicating that the production did occur.

```
30                  | <empty>
31  <id> ::= I-O-CONTROL . <same-list>
32                | <empty>
33  <same-list> ::= <same-element>
34                  | <same-list> <same-element>
35  <same-element> ::= SAME <id-string> .
36  <id-string> ::= <id>
37                  | <id-string> <id>
38  <d-div> ::= DATA DIVISION . <file-section> <work> <link>
39  <file-section> ::= FILE SECTION . <file-list>
```

Actions will differ in production 64 depending upon whether this production has been completed. A flag needs to be set to indicate completion of the file section.

```
40                     | <empty>
```

The flag, indicated in production 39, is set.

```
41  <file-list> ::= <file-element>
42                  | <file-list> <file-element>
43  <files> ::= FD <id> <file-control> . <record-description>
```

This statement indicates the end of a record description, and the length of the record and its address can now be loaded into the symbol table for the file name.

```
44 <file-control> ::= <file-list>
45                   | <empty>
46 <file-list> ::= <file-element>
47                 | <file-list> <file-element>
```

```
48  <file-element> ::= BLOCK <integer> RECORDS
49                  |  RECORD <rec-count>
```

The record length can be saved for comparison with the calculated length from the picture clauses.

```
50                  | LABEL RECORDS STANDARD
51                  | LABEL RECORDS OMITTED
52                  | VALUE OF <id-string>
53  <rec-count> ::= <integer>
54                | <integer> TO <integer>
```

The TO option is the only indication that the file will be variable length. The maximum length must be saved.

```
55  <work> ::= WORKING-STORAGE SECTION . <record-description>
56           | <empty>
57  <link> ::= LINKAGE SECTION . <record-description>
58           | <empty>
59  <record-description> ::= <level-entry>
60                         | <record-descrption> <level-entry>
61  <level-entry> ::= <integer> <data-id> <redefines>
                      <data-type> .
```

The level entry needs to be loaded into the level stack. The level stack is used to keep track of the nesting of field definitions in a record. At this time there may be no information about the length of the item being defined, and its attributes may depend entirely upon its constituent fields. If there is a pending literal, the stack level to which it applies

is saved.

62 <data-id> ::= <id>

63 | FILLER

An entry is built in the symbol table to record information about this record field. It cannot be used explicitly in a program because it has no name, but its attributes will need to be stored as part of the total record.

64 <redefines> ::= REDEFINES <id>

The redefines option gives new attributes to a previously defined record area. The symbol table pointer to the area being redefined is saved so that information can be transfered from one entry to the other. In addition to the information saved relative to the redefinition, it is necessary to check to see if the current level number is less than or equal to the level recorded on the top of the level stack. If this is true, then all information for the item on the top of the stack has been saved and the stack can be reduced.

65 | <empty>

As in production 64, the stack is checked to see if the current level number indicates a reduction of the level stack. In addition, special action needs to be taken if the new level is 01. If an 01 level is encountered at this production prior to production 39 or 40 (the end of the file area), it is an implied

redefinition of the previous 01 level. In the working storage section, it indicates the start of a new record.

```
66  <data-type> ::= <prop-list>
67                | <empty>
68  <prop-list> ::= <data-element>
69                | <prop-list> <data-element>
70  <data-element> ::= PIC <input>
```

The <input> at this point is the character string that defines the record field. It is analyzed and the extracted information is stored in the symbol table.

```
71                   | USAGE COMP
```

The field is defined to be a packed numeric field.

```
72                   | USAGE DISPLAY
```

The DISPLAY format is the default, and thus no special action occurs.

```
73                   | SIGN LEADING <separate>
```

This production indicates the presence of a sign in a numeric field. The sign will be in a leading position. If the <separate> indicator is true, then the length will be one longer than the picture clause, and the type will be changed.

```
74                   | SIGN TRAILING <separate>
```

The same information required by production 73 must be recorded, but in this case the sign is trailing rather than leading.

```
75                   | OCCURS <integer>
```

The type must be set to indicate multiple occurrences, and the number of occurrences saved for computing the space defined by this field.

```
76                    | SYNC <direction>
```

Syncronization with a natural boundary is not required by this machine.

```
77                    | VALUE <literal>
```

The field being defined will be assigned an initial value determined by the value of the literal through the use of an INT operator. This is only valid in the WORKING-STORAGE SECTION.

```
78   <direction> ::= LEFT
79                 | RIGHT
80                 | <empty>
81   <separate> ::= SEPARATE
```

The separate sign indicator is set on.

```
82                | <empty>
83   <literal> ::= <input>
```

The input string is checked to see if it is a valid numeric literal, and if valid, it is stored to be used in a value assignment.

```
84               | <lit>
```

This literal is a quoted string.

```
85               | ZERO
```

As is the case of all literals, the fact that there is a pending literal needs to be saved. In this case and the three following cases, an indicator of which

literal constant is being saved is all that is required. The literal value can be reconstructed later.

86 | SPACE

87 | QUOTE

88 <integer> ::= <input>

The input string is converted to an integer value for later internal use.

89 <id> ::= <input>

The input string is the name of an identifier and is checked against the symbol table. If it is in the symbol table, then a pointer to the entry is saved. If it is not in the symbol table, then an entry is added and the address of that entry is saved.

E. PART TWO

The second part includes all of the PROCEDURE DIVISION, and is the part where code generation takes place. As in the case of the first part, there was no intent to show how various pieces of information were retrieved but only what information was used in producing the output code.

1 <p-div> ::= PROCEDURE DIVISION <using> . <proc-body> EOF

This production indicates termination of the compilation. If the program has sections, then it will be necessary to terminate the last section with a RET 0 instruction. The code will be ended by the output of a TER operation.

2 <using> ::= USING id-string

Not implemented.

3 | <empty>

4 <id-string> ::= <id>

The identifier stack is cleared and the symbol table address of the identifier is loaded into the first stack location.

5 | <id-string> <id>

The identifier stack is incremented and the symbol table pointer stacked.

6 <proc-body> ::= <paragraph>

7 | <proc-body> <paragraph>

8 <paragraph> ::= <id> . <sentence-list>

The starting and ending address of the paragraph are entered into the symbol table. A return is emitted as the last instruction in the paragraph (RET 0). When the label is resolved, it may be necessary to produce a BST operation to resolve previous references to the label.

9 | <id> SECTION .

The starting address for the section is saved. If it is not the first section, then the previous section ending address is loaded and a return (RET 0) is output. As in production 8, a BST may be produced.

10 <sentence-list> ::= <sentence>.

11 | <sentence-list> <sentence> .

12 <sentence> ::= <imperative>

13 | <conditional>

14 | ENTER <id> <opt-id>

This construct is not implemented. An ENTER allows statements from another language to be inserted in the source code.

15 <imperative> ::= ACCEPT <subid>

ACC <address> <length>

16 | <arithmetic>

17 | CALL <lit> <using>

This is not implemented.

18 CLOSE id

CLS file control block address

19 | <file-act>

20 | DISPLAY <lit/id> <opt-lit/id>

The display operator is produced for the first literal or identifier (DIS <address> <length>). If the second value exists, the same code is also produced for it.

21 | EXIT <program-id>

RET 0

22 | GO <id>

BRN <address>

23 | GO <id-string> DEPENDING <id>

GDP is output, followed by a number of parameters: <the number of entries in the identifier stack> <the length of the depending identifier> <the address of

the depending identifier> <the address of each identifier in the stack>.

24 | MOVE <lit/id> TO <subid>

The types of the two fields determine the move that is generated. Numeric moves go through register two using a load and a store. Non-numeric moves depend upon the result field and may be either MOV, MED or MNE. Since all of these instructions have long parameter lists, they have not been listed in detail.

25 | OPEN <type-action> <id>

This produces either OPN, OP1, or OP2 depending upon the <type-action>. Each of these is followed by a file control block address.

26 | PERFORM <id> <thru> <finish>

The PER operation is generated followed by the <branch address> <the address of the return statement to be set> and <the next instruction address>.

27 | <read-id>

28 | STOP <terminate>

If there is a terminate message, then STD is produced followed by <message address> <message length>. Otherwise STP is emitted.

29 <conditional> ::= <arithmetic> <size-error> <imperative>

A BST operator is output to complete the branch around the imperative from production 65.

30 | <file-act> <invalid> <imperative>

A BST operator is output to complete the branch from production 64.

31 | <if-nonterminal> <condition> <action> ELSE <imperative>

NEG will be emitted unless <condition> is a "NOT <cond-type>", in which case the two negatives will cancel each other.

Two BST operators are required. The first fills in the branch to the ELSE action. The second completes the branch around the <imperative> which follows ELSE.

32 | <read-id> <special> <imperative>

A BST is produced to complete the branch around the <imperative>.

33 <Arithmetic> ::= ADD <l/id> <opt-l/id> TO <subid> <round>

The existence of multiple load and store instructions make it difficult to indicate exactly what code will be generated for any of the arithmetic instructions. The type of load and store will depend on the nature of the number involved, and in each case the standard parameters will be produced. This parse step will involve the following actions: first, a load will be emitted for the first number into register zero. If there is a second number, then a load into register one will be produced for it, followed by an ADD and a STI. Next a load into register one will be generated for the result number. Then an ADD instruction will

be emitted. Finally, if the round indicator is set, a RND operator will be produced prior to the store.

34 | DIVIDE <l/id> INTO <subid> <round>

The first number is loaded into register zero. The second operand is loaded into register one. A DIV operator is produced, followed by a RND operator prior to the store, if required.

35 | MULTIPLY <l/id> BY <subid> <round>

The multiply is the same as the divide except that a MUL is produced.

36 | SUBTRACT <l/id> <opt-l/id> FROM <subid> <round>

Subtraction generates the same code as the ADD except that a SUB is produced in place of the last ADD.

37 <file-act> ::= DELETE <id>

Either a DLS or a DLR will be produced along with the required parameters.

38 REWRITE <id>

Either a RWS or a RWR is emitted, followed by parameters.

39 WRITE <id> <special-act>

There are four possible write instructions: WTF, WVL, WRS, and WRR.

40 <condition> ::= <lit/id> <not> <cond-type>

One of the compare instructions is produced. They are CAL, CNS, CNU, RGT, RLT, REQ, SGT, SLT, and SEQ. Two

load instructions and a SUB will also be emitted if one of the register comparisons is required.

```
41  <cond-type> ::= NUMERIC
42                | ALPHABETIC
43                | <compare> <lit/id>
44  <not> ::= NOT
```

NEG is emitted unless the NOT is part of an IF statement in which case the NEG in the IF statement is cancelled.

```
45          | <empty>
46  <compare> ::= GREATER
47              | LESS
48              | EQUAL
49  <ROUND> ::= ROUNDED
50            | <empty>
51  <terminate> ::= <literal>
52                | RUN
53  <special> ::= <invalid>
54              | END
```

An ERO operator is emitted followed by a zero. The zero acts as a filler in the code and will be back-stuffed with a branch address. In this production and several of the following, there is a forward branch on a false condition past an imperative action. For an example of the resolution, examine production 32.

```
55  <opt-id> ::= <subid>
56             |  empty
```

```
57  <action> ::= <imperative>

      BRN 0

58              | NEXT SENTENCE

      BRN 0

59  <thru> ::= THRU <id>

60            |  empty

61  <finish> ::= <1/id> TIMES

      LDI <address> <length> DEC 0

62              | UNTIL <condition>

63              |  empty

64  <invalid> ::= INVALID

      INV 0

65  <size-error> ::= SIZE ERROR

      SER 0

66  <special-act> ::= <when> ADVANCING <how-many>

67                  | <empty>

68  <when> ::= BEFORE

69            | AFTER

70  <how-many> ::= <integer>

71                | PAGE

72  <type-action> ::= INPUT

73                  | OUTPUT

74                  | I-O

75  <subid> ::= <subscript>

76             |  id

77  <integer> ::= <input>
```

The identifier is checked against the symbol table, if it is not present, it is entered as an unresolved label.

```
79  <l/id> ::= <input>
```

The input value may be a numeric literal. If so, it is placed in the constant area with an INT operand. If it is not a numeric literal, then it must be an identifier, and it is located in the symbol table.

```
80         | <subscript>
81         | ZERO
82  <subscript> ::= <id> ( <input> )
```

If the identifier was defined with a USING option, then the input string is checked to see if it is a number or an identifier. If it is an identifier, then an SCR operator is produced.

```
83  <opt-l/id> ::= <l/id>
84                 | <empty>
85  <nn-lit> ::= <lit>
```

The literal string is placed into the constant area using an INT operator.

```
86            | SPACE
87            | QUOTE
88  <literal> ::= <nn-lit>
89              | <input>
```

The input value must be a numeric literal to be valid and is loaded into the constant area using an INT.

```
90              | ZERO
```

```
91  <opt-lit/id> ::= <lit/id>
94                 | <empty>
95  <program-id> ::= <id>
96                 | <empty>
97  <read-id> ::= READ <id>
```

There are four read operations: RDF, RVL, RRS, and RRR.

```
98   <if-nonterminal>::=IF
```

The intermediate code file is the only product of the compiler that is retained. All of the needed information has been extracted from the symbol table, and it is not required by the interpreter. The intermediate code file can be examined through the use of the DECODE Program which translates the output file into a listing of mnemonics followed by the parameters.

APPENDIX A

MICRO-COBOL USER'S MANUAL

TABLE OF CONTENTS

I. ORGANIZATION

The MICRO-COBOL compiler is designed to run on an 8080 system in an interactive mode, and requires at least 16K of RAM memory along with a diskette storage device. The compiler is composed of two parts, each of which reads a portion of the input file. Part one reads the input program and builds the symbol table. At the end of the Data Division, part one is overlayed by part two which uses the symbol table and the Procedure Division of the source program to produce the intermediate code which is written to the diskette as it is generated.

The BUILD Program reads the intermediate code file and creates the executable code memory image which is used by the interpreter. After the memory image has been created, the BUILD Program loads and passes control to the interpreter which then executes the intermediate code.

II. MICRO-COBOL ELEMENTS

The procedure to compile and execute a MICRO-COBOL source program is covered in the next section. This section contains a description of each element in the language and shows simple examples of its use. The following conventions are used in explaining the formats: elements enclosed in broken braces < > are themselves complete entities and are described elsewhere in the manual. Elements enclosed in braces { } are choices, one of the elements which is to be used. Elements enclosed in brackets [ ] are optional. All elements in capital letters are reserved words and must be spelled exactly.

User names are indicated as lower case. These names have been restricted to 12 characters in length. The HYPO-COBOL specification requires that each name start with a letter. There are no restrictions in MICRO-COBOL on what characters must be in any position of a user name. However, it is generally good practice to avoid the use of number strings as names, since they will be taken as literal numbers wherever the context allows it. For example a record could be defined in the Data Division with the name 1234, but the command MOVE 1234 TO RECORD1 would result in the movement of the literal number not the data stored.

The input to the compiler does not need to conform to standard COBOL format. Free form input will be accepted as the default condition. If desired, sequence numbers can be entered in the first six positions of each line. When sequence numbers are used, a compiler parameter must set to cause the compiler to ignore them.

ELEMENT:

IDENTIFICATION DIVISION Format

FORMAT:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. <comment>.
[AUTHOR. <comment>.]
[DATA-WRITTEN. <comment>.]
[SECURITY. <comment>.]
```

DESCRIPTION:

This division provides information for program identification for the reader. The order of the lines is fixed.

EXAMPLES:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.
AUTHOR. PHIL MYLET.
```

ELEMENT:

ENVIRONMENT DIVISION Format

FORMAT:

```
ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. <comment> [DEBUGGING MODE].

OBJECT-COMPUTER. <comment>.

[INPUT-OUTPUT SECTION.

FILE-CONTROL.

   <file-control-entry> . . .

[I-O-CONTROL.

   SAME file-name-1 file-name-2 [file-name-3]

           [file-name-4] [file-name-5].  ]  ]
```

DESCRIPTION:

This division determines the external nature of a file. In the case of CP/M all of the files used can be accessed either sequentially or randomly except for variable length files which are sequential only. The debugging mode is also set by this section.

ELEMENT:

```
     <file-control-entry>
```

FORMAT:

1.

```
     SELECT file-name
        ASSIGN implementor-name
        [ORGANIZATION SEQUENTIAL]
        [ACCESS SEQUENTIAL].
```

2.

```
     SELECT file-name
        ASSIGN implementor-name
        ORGANIZATION RELATIVE
        [ACCESS {SEQUENTIAL [RELATIVE data-name]}].
                {RANDOM RELATIVE data-name      }
```

DESCRIPTION:

The file-control-entry defines the type of file that the program expects to see. There is no difference on the diskette, but the type of reads and writes that are performed will differ. For CP/M the implementor name needs to conform to the normal specifications.

EXAMPLES:

1.

```
     SELECT CARDS
        ASSIGN CARD.FIL.
```

2.

```
SELECT RANDOM-FILE
    ASSIGN A.RAN
    ORGANIZATION RELATIVE
    ACCESS RANDOM RELATIVE RAND-FLAG.
```

ELEMENT:

DATA DIVISION Format

FORMAT:

```
DATA DIVISION.
[FILE SECTION.
[FD file-name
   [BLOCK integer-1 RECORDS]
   [RECORD [integer-2 TO] integer-3]
   [LABEL RECORDS {STANDARD}]
                  {OMITTED}
   [VALUE OF implementor-name-1 literal-1
       [implementor-name-2 literal-2] ... ].
[ record-description-entry ] ...] ...
[WORKING-STORAGE SECTION.
[<record-description-entry>] ... ]
[LINKAGE SECTION.
[<record-description-entry>] ... ]
```

DESCRIPTION:

This is the section that describes how the data is structured. There are no major differences from standard COBOL except for the following: 1. Label records make no sense on the diskette so no entry is required. 2. The VALUE OF clause likewise has no meaning for CP/M. 3. The linkage section has not been implemented.

If a record is given two lengths as in RECORD 12 TO 128, the file is taken to be variable length and can only be accessed in the sequential mode. See the section on files for more information.

ELEMENT:

<comment>

FORMAT:

any string of characters

DESCRIPTION:

A comment is a string of characters. It may include anything other than a period followed by a blank or a reserved word, either of which terminate the string. Comments may be empty if desired, but the terminator is still required by the program.

EXAMPLES:

```
this is a comment
anotheroneallruntogether
8080b 16K
```

ELEMENT:

<data-description-entry> Format

FORMAT:

```
level-number {data-name}
             {FILLER}
[REDEFINES data-name]
[PIC character-string]
[USAGE {COMP}    ]
       {DISPLAY}
[SIGN {LEADING} [SEPARATE]]
      {TRAILING}
[OCCURS integer]
[SYNC [LEFT ]]
      [RIGHT]
[VALUE literal].
```

DESCRIPTION:

This statement describes the specific attributes of the data. Since the 8080 is a byte machine, there was no meaning to the SYNC clause, and thus it has not been implemented.

EXAMPLES:

```
      01 CARD-RECORD.
         02 PART PIC X(5).
         02 NEXT-PART PIC 99V99 USAGE COMP.
         02 FILLER.
            03 NUMB PIC S9(3)V9 SIGN LEADING SEPARATE.
            03 LONG-NUMB 9(15).
            03 STRING REDEFINES LONG-NUMB PIC X(15).
         02 ARRAY PIC 99 OCCURS 100.
```

ELEMENT:

PROCEDURE DIVISION Format

FORMAT:

1.

```
PROCEDURE DIVISION [USING namel [name2] ... [name5]].
section-name SECTION.
[paragraph-name. <sentence> [<sentence> ... ] ... ] ...
```

2.

```
PROCEDURE DIVISION [USING namel [name2] ... [name5].
paragraph-name. <sentence> [<sentence> ...] ...
```

DESCRIPTION:

As is indicated, if the program is to contain sections, then the first paragraph must be in a section. The USING option is part of the inter-program communication module and has not been implemented.

ELEMENT:

<sentence>

FORMAT:

<imperative-statement>

<conditional-statement>

ENTER verb

DESCRIPTION:

All sentences other than ENTER fall in one of the two main categories. ENTER is part of the inter-program communication module.

<imperative-statement>

ELEMENT:

<imperative-statement>

FORMAT

The following verbs are always imperatives:

ACCEPT

CALL

CLOSE

DISPLAY

EXIT

GO

MOVE

OPEN

PERFORM

STOP

The following may be imperatives:

arithmetic verbs without the SIZE ERROR statement and DELETE, WRITE, and REWRITE without the INVALID option.

ELEMENT:

<conditional-statements>

FORMAT:

IF

READ

arithmetic verbs with the SIZE ERROR statement

and DELETE, WRITE, and REWRITE with the INVALID

option.

ELEMENT:

ACCEPT

FORMAT:

ACCEPT <identifier>

DESCRIPTION:

This statement reads up to 72 characters from the console. The usage of the item must be DISPLAY.

EXAMPLES:

```
ACCEPT IMMAGE

ACCEPT NUM(9)
```

ELEMENT:

ADD

FORMAT:

```
ADD {identifier} [{identifier-1}] TO identifier-2
    {literal}     {literal     }
   [ROUNDED] [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

This instruction adds either one or two numbers to a third with the result being placed in the last location.

EXAMPLES:

```
ADD 10 TO NUMB1
ADD X Y TO Z ROUNDED.
ADD 100 TO NUMBER SIZE ERROR GO ERROR-LOC
```

ELEMENT:

CALL

FORMAT:

```
CALL literal [USING name1 [name2] ... [name5]
```

DESCRIPTION:

CALL is not implemented.

ELEMENT:

```
CLOSE
```

FORMAT:

```
CLOSE file-name
```

DESCRIPTION:

Files must be closed if they have been written. However, the normal requirement to close an input file prior to the end of processing does not exist.

EXAMPLES:

```
CLOSE FILE1
CLOSE RANDFILE
```

ELEMENT:

DELETE

FORMAT:

```
DELETE record-name [INVALID <imperative-statement>]
```

DESCRIPTION:

This statement requires the record name, not the file name as in the standard form of the statement. Since there is no deletion mark in CP/M, this would normally result in the record still being readable. It is, therefore, filled with zeroes to indicate that it has been removed.

EXAMPLES:

```
DELETE RECORD1
```

ELEMENT:

DISPLAY

FORMAT:

```
DISPLAY {identifier} [{identifier-1}]
        {literal   }  {literal     }
```

DESCRIPTION:

This displays the contents of an identifier or displays a literal on the console. Usage must be DISPLAY. The maximum length of the display is 72 positions.

EXAMPLES:

```
DISPLAY MESSAGE-1
DISPLAY MESSAGE-3 10
DISPLAY 'THIS MUST BE THE END'
```

ELEMENT:

DIVIDE

FORMAT:

```
DIVIDE {identifier} INTO identifier-1 [ROUNDED]
       {literal   }
   [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

The result of the division is stored in identifier-1; any remainder is lost.

EXAMPLES:

```
DIVIDE NUMB INTO STORE
DIVIDE 25 INTO RESULT
```

ELEMENT:

EXIT

FORMAT:

```
EXIT [PROGRAM]
```

DESCRIPTION:

The EXIT command causes no action by the interpreter but allows for an empty paragraph for the construction of a common return point. The optional PROGRAM statement is not implemented as it is part of the inter-program communication module.

EXAMPLES:

```
RETURN.
EXIT.
```

ELEMENT:

GO

FORMAT:

1.

```
GO procedure-name
```

2.

```
GO procedure-1 [procedure-2] ... procedure-20
   DEPENDING identifier
```

DESCRIPTION:

The GO command causes an unconditional branch to the routine specified. The second form causes a forward branch depending on the value of the contents of the identifier. The identifier must be a numeric integer value. There can be no more than 20 procedure names.

EXAMPLES:

```
GO READ-CARD.
GO READ1 READ2 READ3 DEPENDING READ-INDEX.
```

ELEMENT:

IF

FORMAT:

```
IF <condition> {imperative}     ELSE imperative-2
               {NEXT SENTENCE}
```

DESCRIPTION:

This is the standard COBOL IF statement. Note that there is no nesting of IF statements allowed since the IF statement is a conditional.

EXAMPLES:

```
IF A GREATER B ADD A TO C ELSE GO ERROR-ONE.
IF A NOT NUMERIC NEXT SENTENCE ELSE MOVE ZERO TO A.
```

ELEMENT:

MOVE

FORMAT:

```
MOVE {identifier-1} TO identifier-2
     {literal     }
```

DESCRIPTION:

The standard list of allowable moves applies to this action. As a space saving feature of this implementation, all numeric moves go through the accumulators. This makes numeric moves slower than alphanumeric moves, and where possible they should be avoided. Any move that involves picture clauses that are exactly the same can be accomplished as an alphanumeric move if the elements are redefined as alphanumeric; also all group moves are alphanumeric.

EXAMPLES:

```
MOVE SPACE TO PRINT-LINE.
MOVE A(10) TO B(PTR).
```

ELEMENT:

MULTIPLY

FORMAT:

```
MULTIPLY {identifier} BY identifier-2 [ROUNDED]
         {literal   }
    [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

The multiply routine requires enough space to calculate the result with the full number of decimal digits prior to moving the result into identifier-2. This means that a number with 5 places after the decimal multiplied by a number with 6 places after the decimal will generate a number with 11 decimal places which would overflow if there were more than 7 digits before the decimal place.

EXAMPLES:

```
MULTIPLY X BY Y.
MULTIPLY A BY B(7) SIZE ERROR GO OVERFLOW.
```

ELEMENT:

OPEN

FORMAT:

```
OPEN {INPUT file-name }
     {OUTPUT file-name}
     {I-O file-name   }
```

DESCRIPTION:

All three types of OPENs have the same effect on the diskette. However, they do allow for internal checking of the other file actions. For example, a write to a file set open as input will cause a fatal error.

EXAMPLES:

```
OPEN INPUT CARDS.
OPEN OUTPUT REPORT-FILE.
```

ELEMENT:

PERFORM

FORMAT

1.

```
PERFORM procedure-name [THRU procedure-name-2]
```

2.

```
PERFORM procedure-name [THRU procedure-name-2]
   {identifier} TIMES
   {integer   }
```

3.

```
PERFORM procedure-name [THRU procedure-name-2]
   UNTIL <condition>
```

DESCRIPTION:

All three options are supported. Branching may be either forward or backward, and the procedures called may have perform statements in them as long as the end points do not coincide or overlap.

EXAMPLES:

```
PERFORM OPEN-ROUTINE.
PERFORM TOTALS THRU END-REPORT.
PERFORM SUM 10 TIMES.
PERFORM SKIP-LINE UNTIL PG-CNT GREATER 60.
```

ELEMENT:

READ

FORMAT:

1.

```
READ file-name INVALID <imperative-statement>
```

2.

```
READ file-name END <imperative-statement>
```

DESCRIPTION:

The invalid condition is only applicable to files in a random mode. All sequential files must have an END statement.

EXAMPLES:

```
READ CARDS END GO END-OF-FILE.
READ RANDOM-FILE INVALID MOVE SPACES TO REC-1.
```

ELEMENT:

REWRITE

FORMAT:

```
REWRITE file-name [INVALID <imperative>]
```

DESCRIPTION:

REWRITE is only valid for files that are open in the 1-0 mode. The INVALID clause is only valid for random files. This statement results in the current record being written back into the place that it was just read from. Note that this requires a file name not a record name.

EXAMPLES:

```
REWRITE CARDS.
REWRITE RAND-1 INVALID PERFORM ERROR-CHECK.
```

ELEMENT:

STOP

FORMAT:

```
STOP {RUN    }
     {literal}
```

DESCRIPTION:

This statement ends the running of the interpreter. If a literal is specified, then the literal is displayed on the console prior to termination of the program.

EXAMPLES:

```
STOP RUN.
STOP 1.
STOP "INVALID FINISH".
```

ELEMENT:

SUBTRACT

FORMAT:

```
SUBTRACT {identifier-1} [identifier-2] FROM identifer-3
         {literal-1   } [literal-2   ]
   [ROUNDED] [SIZE ERROR <imperative-statement>]
```

DESCRIPTION:

Identifier-3 is decremented by the value of identifier/literal one, and, if specified, identifier/literal two. The results are stored back in identifier-3. Rounding and size error options are available if desired.

EXAMPLES:

```
SUBTRACT 10 FROM SUB(12).
SUBTRACT A B FROM C ROUNDED.
```

ELEMENT:

WRITE

FORMAT:

1.

```
WRITE file-name [{BEFORE} ADVANCING {INTEGER}]
                 {AFTER }           {PAGE   }
```

2.

```
WRITE file-name INVALID <imperative-statement>
```

DESCRIPTION:

There is no printer on the 8080 system here, so the ADVANCING option is not implemented. The INVALID option only applies to random files.

EXAMPLES:

```
WRITE OUT-FILE.
WRITE RAND-FILE INVALID PERFORM ERROR-RECOV.
```

ELEMENT:

```
      <condition>
```

FORMAT:

```
      RELATIONAL CONDITION:

         {identifier-1} [NOT] {GREATER} {identifier-2}
         {literal-1   }       {LESS   } {literal-2   }
                              {EQUAL  }

      CLASS CONDITION:

         identifier [NOT] {NUMERIC   }
                          {ALPHABETIC}
```

DESCRIPTION:

It is not valid to compare two literals. The class condition NUMERIC will allow for a sign if the identifier is signed numeric.

EXAMPLES:

```
      A NOT LESS 10.

      LINE GREATER "C".

      NUMB1 NOT NUMERIC.
```

ELEMENT:

Subscripting

FORMAT:

```
data-name (subscript)
```

DESCRIPTION:

Any item defined with an OCCURS may by referenced by a subscript. The subscript may be a literal integer, or it may be a data item that has been specified as an integer. If the subscript is signed, the sign must be positive at the time of its use.

EXAMPLES:

```
A(10)

ITEM(SUB)
```

III. COMPILER PARAMETERS

There are four compiler parameters which are controlled by entries on the first line of the program. A parameter consists of a dollar sign followed by a letter.

$L -- list the input code on the screen as the program is compiled. Default is on. Error messages will be difficult to understand with this parameter turned off, but it may be desirable when used witha slow output device.

$S -- sequence numbers are in the first six positions of each record. Default is off.

$P -- list productions as they occur. Default is off.

$T -- list tokens from the scanner. Default is off.

IV. RUN TIME CONVENTIONS

This section explains how to compile and execute MICRO-COBOL source programs. The compiler expects to see a file with a type of CBL as the input file. In general, the input is free form. If the input includes line numbers then the compiler must be notified by setting the appropriate parameter. The compiler is started by typing COBOL <file-name>. Where the file name is the system name of the input file. There is no interaction required to start the second part of the compiler. The output file will have the same file name as the input file, and will be given a file type of CIN. Any previous copies of the file will be erased.

The interpreter is started by typing EXEC <file-name>. The first program is a loader, and it will display "LOAD FINISHED" to indicate successful completion. The run-time package will be brought in by the build program, and execution should continue without interruption.

V. FILE INTERACTIONS WITH CP/M

The file structure that is expected by the program imposes some restrictions on the system. References 3 and 4 contain detailed information on the facilities of CP/M, and should be consulted for details. The information that has been included in this section is intended to explain where limitations exist and how the program interacts with the system.

All files in CP/M are on a random access device, and there is no way for the system to distinguish sequential files from files created in a random mode. This means that the various types of reads and writes are all valid to any file that has fixed length records. The restrictions of the ASSIGN statement do prevent a file from being open for both random and sequential actions during one program.

Each logical record is terminated by a carriage return and a line feed. In the case of variable length records, this is the only end mark that exists. This convention was adopted to allow the various programs which are used in CP/M to work with the files. Files created by the editor, for example, will generally be variable length files. This convention does remove the capability of reading variable length files in a random mode.

All of the physical records are assumed to be 128 bytes in length, and the program supplies buffer space for

records in addition to the logical records. Logical records may be of any desired length.

VI. ERROR MESSAGES

A. COMPILER FATAL MESSAGES

BR	Bad read -- disk error, no corrective action can be taken in the program.
CL	Close error -- unable to close the output file.
MA	Make error -- could not create the output file.
MO	Memory overflow -- the code and constants generated will not fit in the alloted memory space.
OP	Open error -- can not open the input file, or no such file present.
ST	Symbol table overflow -- symbol table is too large for the allocated space.
WR	Write error -- disk error, could not write a code record to the disk.

B. COMPILER WARNINGS

EL	Extra levels -- only 10 levels are allowed.
FT	File type -- the data element used in a read or write statement is not a file name.
IA	Invalid access -- the specified options are not an allowable combination.

ID	Identifier stack overflow -- more than 20 items in a GO TO -- DEPENDING statement.
IS	Invalid subscript -- an item was subscripted but it was not defined by an OCCURS.
IT	Invalid type -- the field types do not match for this statement.
LE	Literal error -- a literal value was assigned to an item that is part of a group item previously assigned a value.
NF	No file assigned -- there was no SELECT clause for this file.
NI	Not implemented -- a production was used that is not implemented.
NN	Non-numeric -- an invalid character was found in a numeric string.
NP	No production -- no production exists for the current parser configuration; error recovery will automatically occur.
NV	Numeric value -- a numeric value was assigned to a non-numeric item.
PC	Picture clause -- an invalid character or set of characters exists in the picture clause.

PF	Paragraph first -- a section header was produced after a paragraph header, which is not in a section.
R1	Redefine nesting -- a redefinition was made for an item which is part of a redefined item.
R2	Redefine length -- the length of the redefinition item was greater than the item that it redefined.
SE	Scanner error -- the scanner was unable to read an identifier due to an invalid character.
SG	Sign error -- either a sign was expected and not found, or a sign was present when not valid.
SL	Significance loss -- the number assigned as a value is larger than the field defined.
TE	Type error -- the type of a subscript index is not integer numeric.
VE	Value error -- a value statement was assigned to an item in the file section.

C. INTERPRETER FATAL ERRORS

CL	Close error -- the system was unable to close an output file.
ME	Make error - the system was unable to make an input file on the disk.

NF	No file -- an input file could not be opened.
WI	Write to input -- a write was attempted to an input file.

D. INTERPRETER WARNING MESSAGES

EM	End mark -- a record that was read did not have a carriage return or a line feed in the expected location.
GD	Go to depending -- the value of the depending indicator was greater than the number of available branch addresses.
IC	Invalid character -- an invalid character was loaded into an output field during an edited move. For example, a numeric character into an alphabetic-only field.
SI	Sign invalid -- the sign is not a "+" or a "-".

wR.

LIST OF REFERENCES

1. Mylet, P. R. MICRO-COBOL a subset of Navy Standard HYPO-COBOL for Micro-computers, Master's Thesis; Naval Postgraduate School, September 1978.

2. Craig, A. S. MICRO-COBOL an implementation of Navy Standard HYPO-COBOL for microprocessor-based computer systems, Master's Thesis, Naval Postgraduate School, March 1977.

3. Digital Research, An Introduction to CP/M Features and Facilities, 1976.

4. Digital Research, CP/M Interface Guide, 1976.

5. Intel Corporation, 8008 and 8080 PL/M Programming Manual, 1975.

6. Intel Corporation, 8080 Simulator Software Package, 1974.

7. Software Development Division, ADPE Selection Office, Department of the Navy, HYPO-COBOL, April 1975.

APPENDIX B

MICRO-COBOL FILE CREATION

The MICRO-COBOL compiler and interpreter source files currently exist in PLM80 and are edited and compiled under ISIS on the INTEL MDS System. This is a description of the procedure used to create the executable files required to compile and interpret MICRO-COBOL programs. The MICRO-COBOL compiler and interpreter run under CP/M by executing the following four object code files.

1. COBOL.COM
2. PART2.COM
3. EXEC.COM
4. INTERP.COM

These four files are created from the following six PLM80 source programs.

1. PART1.PLM
2. PART2.PLM
3. BUILD.PLM
4. INTERP.PLM
5. INTRDR.PLM
6. READER.PLM

The procedure used to create the four object files involves compiling, linking, and locating each of the six source files under ISIS. The DDT program is then used under CP/M to construct the executable files. Each of the

following steps describe the action to be taken and, where appropriate, the command string to be entered into the computer.

1. An ISIS system diskette containing the PLM80 compiler is placed into drive A and a non-system diskette containing the source programs is placed into drive B.

2. Compile the PLM source file under ISIS.

```
PLM80 :F1:<filename>.PLM DEBUG
```

DEBUG saves the symbol table and line files for later use during debugging sessions.

3. Link the PLM80 object file.

```
LINK   :F1:<filename>.OBJ,   TRINT.OBJ,   PLM80.LIB   TO
:F1:<filename>.MOD
```

4. Locate object file.

```
LOCATE :F1:<filename>.MOD CODE(103H)
```

5. Replace ISIS system diskette in drive A with a CP/M system diskette and reboot the system.

6. Transfer the located ISIS file from the diskette in drive B to the CP/M diskette in drive A.

```
FROMISIS <filename>
```

7. Convert the ISIS file to CP/M executable form.

```
OBJCPM <filename>
```

At this point the object file is in machine readable form and will run under CP/M when called properly. INTERP.COM and PART2.COM are called by EXEC.COM and PART1.COM and need no further work. EXEC.COM and PART1.COM need to be constructed from the remaining four files.

EXEC.COM is created by entering the following commands under CP/M.

1. DDT BUILD.COM
2. IINTRDR.HEX
3. R1C00
4. A1CB5
5. JMP 5
6. A1CC1
7. JMP 5
8. CONTROL-C
9. SAVE 29 EXEC.COM

PART1.COM is created by entering the following commands under CP/M.

1. DDT PART1.COM
2. IREADER.HEX
3. RFB00 6200
4. A1F90
5. JMP 3100 0P[illegible]0
6. Control-C
7. SAVE 44 COBOL.COM

MICRO-COBOL programs may now be executed in the following manner. The source program is named, <filename>.CBL. The command, "COBOL <filename>", causes the MICRO-COBOL source program, <filename>.CBL, to be read in from diskette and compiled. During the compile, the intermediate code file, <filename>.CIN, is written out to diskette as it is generated. The command, "EXEC <filename>", causes the file, <filename>.CIN, to be executed.

APPENDIX C

LIST OF INOPERATIVE CONSTRUCTS

The following is a list of MICRO-COBOL elements that were not implemented at the beginning of this project. In most cases code had been written to implement the element but is was either incomplete or incorrect. The elements marked with an asterisk still have bugs and need additional work.

MULTIPLY

<condition>

STOP <literal>

IF

PERFORM <procedure 1> THRU <procedure 2>

PERFORM <procedure> <n> TIMES

PERFORM <procedure> UNTIL <condition>

FILE I/O *

Numeric Edit *

The following HYPO-COBOL elements are part of MICRO-COBOL only to the extent that they are defined in the grammar. No code has been written to support them.

USING

CALL

ENTER

<when> ADVANCING <how-many>

It must be pointed out that this information is based only on informal testing with very simple programs. MICRO-COBOL is only now at a stage at which it is appropriate to conduct exhaustive testing using the HYPO-COBOL Compiler Validation System.

APPENDIX D

MICRO-COBOL PARSE TABLE GENERATION

The parse tables for MICRO-COBOL were generated on the IBM 360 using the LALR(1) parse table generator described in Reference 11. There are basically two steps involved in generating the tables. First, a deck of cards containing the grammar is entered into the computer using the following JCL:

```
//GO EXEC PGM= LALR,REGION=220K
//STEPLIB DD DSN-F0963.LALR,UNIT=2314,
      VOL=SER=LINDA,DISP=SHR
//SYSPRINT DD SYSOUT=A,DCB=(RECFM=FB,
      LRECL=133,BLKSIZE=3325),
//SPACE=(CYL,(1,1))
//NONTERM DD SPACE=(CYL,(1,1)),UNIT=SYSDA
//FSMDATA DD SPACE=(CYL,(1,1)),UNIT=SYSDA
//PTABLES DD SYSOUT=B,
      DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
//SYSIN DD *
```

The output from this run is a listing and deck containing the tables in XPL compatible format. This deck is then translated into PLM compatible format using the following JCL and an XPL program which is available in the card deck library in the Computer Science Department at the Naval Postgraduate School.

```
    //EXEC XCOM
    //COMP.SYSIN DD *
    //GO.SYSPUNCH DD SYSOUT=B,
          DCB=(RECFM=FB,LRECL=80,BLKSIZE=800)
    //GO.SYSIN DD *
```

The tables are then transferred to a diskette and edited into the PLM80 source program using the ISIS COPY and EDIT features on the INTEL MDS System.

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE READER
OBJECT MODULE PLACED IN :F1:READER.OBJ
COMPILER INVOKED BY:  PLM80 :F1:READER.PLM


              $ PAGELENGTH(90)
   1          READER:
                    DO;

                  /* COBOL COMPILER - PART 2 READER */

                  /* THIS PROGRAM IS LOADED IN WITH THE PART 1 PROGRAM
                  AND IS CALLED WHEN PART 1 IS FINISHED.  THIS PROGRAM
                  OPENS THE PART2.COM FILE THAT CONTAINS THE CODE FOR
                  PART 2 OF THE COMPILER, AND READS IT INTO CORE.  AT
                  THE END OF THE READ OPERATION, CONTROL IS PASSED TO
                  THE SECOND PART PROGRAM                                    */


              /*           3100H:     LOAD POINT */

   2   1      DECLARE

              START    LITERALLY '100H',   /* STARTING LOCATION FOR PASS 2 */
              ADR      ADDRESS INITIAL(START),
              FCB (33) BYTE INITIAL(0,'PASS2   COM',0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0),
              I        ADDRESS;

   3   1      MON1: PROCEDURE(F,A) EXTERNAL;
   4   2          DECLARE F BYTE, A ADDRESS;
   5   2      END MON1;

   6   1      MON2: PROCEDURE(F,A)BYTE EXTERNAL;
   7   2          DECLARE F BYTE, A ADDRESS;
   8   2      END MON2;

   9   1      BOOT: PROCEDURE EXTERNAL;
  10   2          END;

  11   1      OPEN: PROCEDURE (FCB) BYTE;
  12   2            DECLARE FCB ADDRESS;
  13   2            RETURN MON2 (15, FCB);
  14   2            END;

  15   1      READ: PROCEDURE (ADDR) BYTE;
  16   2            DECLARE ADDR ADDRESS;
  17   2            CALL MON1 (26, ADDR);   /* SET DMA ADDRESS */
  18   2            RETURN MON2 (20,  FCB);   /*  READ, AND RETURN ERROR CODE   */
  19   2            END;

  20   1      ERROR: PROCEDURE(CODE);
  21   2          DECLARE CODE ADDRESS;
  22   2          CALL MON1(2,(HIGH(CODE)));
  23   2          CALL MON1(2,(LOW(CODE)));
  24   2          CALL TIME(10);
  25   2          CALL BOOT;
  26   2      END ERROR;
  27   1          CALL MON1 (26, 0100H);

                   /* OPEN PASS2.COM */
  28   1      IF OPEN(.FCB)=255 THEN CALL ERROR('02');
                  /* READ IN FILE */

  30   1            I = 0100H;    /* INITIAL ADDRESS   */
  31   1            DO WHILE READ(I) = 0;       /* READ 1 SECTOR */
  32   2                I = I + 0080H;        /* BUMP DMA ADDRESS   */
  33   2            END;

  34   1           CALL MON1 (26, 0080H);    /*  RESET DMA ADDRESS   */
  35   1            CALL ADR;

  36   1         END;


MODULE INFORMATION:

     CODE AREA SIZE     = 009DH    157D
     VARIABLE AREA SIZE = 002BH     43D
     MAXIMUM STACK SIZE = 0004H      4D
     67 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INTRDR
OBJECT MODULE PLACED IN :F1:INTRDR.OBJ
COMPILER INVOKED BY:  PLM80 :F1:INTRDR.PLM


               $PAGELENGTH(90)
   1           INTRDR:         /*  NAME OF MODULE   */
                     DO;

                  /* COBOL COMPILER - INTERP READER */

                  /* THIS PROGRAM IS CALLED BY THE BUILD PROGRAM AFTER
                  CINTERP.COM HAS BEEN OPENED, AND READS THE CODE INTO MEMORY
                  */


                  /*    80H  -  LOAD POINT       */

   2   1       DECLARE

               START     LITERALLY '100H',   /* STARTING LOCATION FOR PASS 2 */
               INTERP    ADDRESS  INITIAL(START),
               I    ADDRESS  INITIAL (0080H);

   3   1       MONA  PROCEDURE(F,A);
   4   2           DECLARE F BYTE, A ADDRESS;
   5   2         L: GO TO L;    /*    PATCH TO ->  "JMP  BDOS"     */
   6   2       END MONA;

   7   1       MONB  PROCEDURE(F,A)BYTE;
   8   2           DECLARE F BYTE, A ADDRESS;
   9   2         L: GO TO L;    /*    PATCH TO ->  "JMP  BDOS"     */
  10   2           RETURN 0;        /*    ZAP -> "NO-OP"    */
  11   2       END MONB;

  12   1           DO WHILE 1;
  13   2               CALL MONA (26, (I:=I+0080H));    /* SET DMA ADDRESS */
  14   2               IF   MONB (20, 5CH) <> 0 THEN
  15   2                                                   CALL INTERP;
  16   2           END;
  17   1       END;


MODULE INFORMATION:

     CODE AREA SIZE      = 0047H     71D
     VARIABLE AREA SIZE  = 000AH     10D
     MAXIMUM STACK SIZE  = 0002H      2D
     36 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
ISIS-II PL/M-80 V3 1 COMPILATION OF MODULE BUILD
OBJECT MODULE PLACED IN  F1 BUILD OBJ
COMPILER INVOKED BY   PLM80  F1 BUILD PLM


             $PAGELENGTH(90)
             BUILD
             DO,
             /* NORMALLY ORG ED AT 100H */

                 /* THIS PROGRAM TAKES THE CODE OUTPUT FROM THE COBOL COMPILER
                 AND BUILDS THE ENVIRONMENT FOR THE COBOL INTERPRETER */


             DECLARE

             LIT             LITERALLY        'LITERALLY',
             BOOT            LIT              '0',
             BDOS            LIT              '5',
             TRUE            LIT              '1',
             FALSE           LIT              '0',
             FOREVER         LIT              'WHILE TRUE',
             FCB             ADDRESS          INITIAL (5CH),
             FCB$BYTE        BASED      FCB   BYTE,
             FCB$BYTE$A      BASED FCB (33)   BYTE,
             I               BYTE,
             ADDR            ADDRESS          INITIAL (100H),
             CHAR            BASED      ADDR  BYTE,
             BUFF$END        LIT               100H',
             INTERP$FCB      (33)       BYTE  INITIAL(0, 'CINTERP COM',0,0,0,0),
             CODE$NOT$SET  BYTE         INITIAL (TRUE),
             READER$LOCATION     LIT           1C80H ,
             INTERP$ADDRESS      ADDRESS      INITIAL(2000H),
             INTERP$CONTENT      BASED        INTERP$ADDRESS ADDRESS,
             I$BYTE              BASED        INTERP$ADDRESS (2) BYTE,
             CODE$CTR            ADDRESS,
             C$BYTE              BASED        CODE$CTR  BYTE,
             BASE                ADDRESS,
             B$ADDR              BASED         BASE      ADDRESS,
             B$BYTE              BASED         BASE (4) BYTE,

             MON1. PROCEDURE (F,A) EXTERNAL,
                 DECLARE F BYTE, A ADDRESS,
             END MON1,


             MON2  PROCEDURE (F,A) BYTE EXTERNAL,
                 DECLARE F BYTE, A ADDRESS,
             END MON2,


             PRINT$CHAR  PROCEDURE(CHAR),
                 DECLARE CHAR BYTE,
                 CALL MON1(2,CHAR),
             END PRINT$CHAR,


             CRLF  PROCEDURE,
                 CALL PRINT$CHAR(13),
                 CALL PRINT$CHAR(10),
             END CRLF,


             PRINT  PROCEDURE(A),
                 DECLARE A ADDRESS,
                 CALL CRLF,
                 CALL MON1(9,A),
             END PRINT,


             OPEN  PROCEDURE (A) BYTE,
                 DECLARE A ADDRESS,
                 RETURN MON2(15,A),
             END OPEN,

             REBOOT  PROCEDURE,
                 ADDR = BOOT, CALL ADDR,
                 END REBOOT,


             MOVE  PROCEDURE(FROM, DEST, COUNT),
                 DECLARE (FROM, DEST, COUNT) ADDRESS,
                        (F BASED FROM, D BASED DEST) BYTE,
                 DO WHILE(COUNT =COUNT-1)<>0FFFFH,
                        D=F,
                        FROM=FROM+1,
```

```
 3                 DEST=DEST+1;
 3            END;
 2       END MOVE;


 1       GET$CHAR: PROCEDURE BYTE;
 2            IF (ADDR =ADDR + 1)>=BUFF$END THEN
 2            DO;
 3                 IF MON2(20,FCB)<>0 THEN
 3                 DO;
 4                      CALL PRINT(.('END OF INPUT  $'));
 4                      CALL REBOOT;
 4                 END;
 3                 ADDR=80H;
 3            END;
 2            RETURN CHAR;
 2       END GET$CHAR;


 1       NEXT$CHAR: PROCEDURE;
 2            CHAR=GET$CHAR;
 2       END NEXT$CHAR;


 1       STORE: PROCEDURE(COUNT);
 2            DECLARE COUNT BYTE;
 2            IF CODE$NOT$SET THEN
 2            DO;
 3                 CALL PRINT(.('CODE ERROR$ '));
 3                 CALL NEXT$CHAR;
 3                 RETURN;
 3            END;
 2            DO I=1 TO COUNT;
 3                 C$BYTE=CHAR;
 3                 CALL NEXT$CHAR;
 3                 CODE$CTR=CODE$CTR+1;
 3            END;
 2       END STORE;


 1       BACK$STUFF: PROCEDURE;
 2            DECLARE (HOLD,STUFF) ADDRESS;
 2            BASE= HOLD;
 2            DO I=0 TO 3;
 3                 B$BYTE(I)=GET$CHAR;
 3            END;
 2            DO FOREVER;
 3                 BASE=HOLD;
 3                 HOLD=B$ADDR;
 3                 B$ADDR=STUFF;
 3                 IF HOLD=0 THEN
 3                 DO;
 4                      CALL NEXT$CHAR;
 4                      RETURN;
 4                 END;
 3            END;
 2       END BACK$STUFF;


 1       START$CODE: PROCEDURE;
 2            CODE$NOT$SET=FALSE;
 2            I$BYTE(0)=GET$CHAR;
 2            I$BYTE(1)=GET$CHAR;
 2            CODE$CTR=INTERP$CONTENT;
 2            CALL NEXT$CHAR;
 2       END START$CODE;


 1       GO$DEPENDING: PROCEDURE;
 2            CALL STORE(1);
 2            CALL STORE(SHL(CHAR,1) + 4);
 2       END GO$DEPENDING;


 1       INITIALIZE: PROCEDURE;
 2            DECLARE (COUNT,WHERE,HOW$MANY) ADDRESS;
 2            BASE= WHERE;
 2            DO I=0 TO 3;
 3                 B$BYTE(I)=GET$CHAR;
 3            END;
 2            BASE=WHERE - 1;
 2            DO COUNT = 1 TO HOW$MANY;
 3                 B$BYTE(COUNT)=GET$CHAR;
 3            END;
 2            CALL NEXT$CHAR;
 2       END INITIALIZE;
```

```
BUILD: PROCEDURE;
    DECLARE
    F2   LIT  '9',
    F3   LIT  '9',
    F4   LIT  '21',
    F5   LIT  '24',
    F6   LIT  '32',
    F7   LIT  '39',
    F9   LIT  '49',
    F10  LIT  '54',
    F11  LIT  '60',
    F13  LIT  '61',
    GDP  LIT  '62',
    INT  LIT  '63',
    BST  LIT  '64',
    TER  LIT  '65',
    SCO  LIT  '66';

    DO FOREVER;
        IF CHAR < F2 THEN CALL STORE(1);
        ELSE IF CHAR < F3 THEN CALL STORE(2);
        ELSE IF CHAR < F4 THEN CALL STORE(3);
        ELSE IF CHAR < F5 THEN CALL STORE(4);
        ELSE IF CHAR < F6 THEN CALL STORE(5);
        ELSE IF CHAR < F7 THEN CALL STORE(6);
        ELSE IF CHAR < F9 THEN CALL STORE(7);
        ELSE IF CHAR < F10 THEN CALL STORE(9);
        ELSE IF CHAR < F11 THEN CALL STORE(10);
        ELSE IF CHAR < F13 THEN CALL STORE(11);
        ELSE IF CHAR < GDP THEN CALL STORE(13);
        ELSE IF CHAR = GDP THEN CALL GO$DEPENDING;
        ELSE IF CHAR = BST THEN CALL BACK$STUFF;
        ELSE IF CHAR = INT THEN CALL INITIALIZE;
        ELSE IF CHAR = TER THEN
        DO;
            CALL PRINT(.('LOAD FINISHED$'));
            RETURN;
        END;
        ELSE IF CHAR = SCO THEN CALL START$CODE;
        ELSE DO;
            IF CHAR <> 0FFH THEN CALL PRINT(.('LOAD ERROR$'));
            CALL NEXT$CHAR;
        END;
    END;
END BUILD;

    /* PROGRAM EXECUTION STARTS HERE */

FCB$BYTE$A(32), FCB$BYTE=0;
CALL MOVE(.('CIN',0,0,0,0),FCB + 9,7);
IF OPEN(FCB)=255 THEN
DO;
    CALL PRINT(.('FILE NOT FOUND    $'));
    CALL REBOOT;
END;
CALL NEXT$CHAR;
CALL BUILD;
CALL MOVE(.INTERP$FCB,FCB,33);
FCB$BYTE$A(32) = 0;
IF OPEN(FCB)=255 THEN
DO;
    CALL PRINT(.('INTERPRETER NOT FOUND    $'));
    CALL REBOOT;
END;
CALL MOVE(READER$LOCATION, 80H, 80H);
ADDR = 80H; CALL ADDR; /* BRANCH TO 80H */
END;


MODULE INFORMATION

     CODE AREA SIZE     = 0402H    1026D
     VARIABLE AREA SIZE = 0043H      67D
     MAXIMUM STACK SIZE = 0012H      18D
     237 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE PART1
OBJECT MODULE PLACED IN  F1 PART1.OBJ
COMPILER INVOKED BY   PLM80  F1 PART1.PLM



               $PAGELENGTH(90)
  1            PART1
               DO;
               /* NORMALLY ORG'ED AT 100H */

                   /*      COBOL COMPILER - PART 1              */



                      /*    GLOBAL DECLARATIONS AND LITERALS    */

  2   1        DECLARE LIT LITERALLY 'LITERALLY',
  3   1        DECLARE
                   MAX$MEMORY     LIT        '3100H',   /* TOP OF USEABLE MEMORY */
                   INITIAL$POS    LIT        '2C00H',
                   RDR$LENGTH     LIT        '255',
                   PASS1$LEN      LIT        '46',
                   CR             LIT        '13',
                   LF             LIT        '10',
                   QUOTE          LIT        '22H',
                   POUND          LIT        '23H',
                   TRUE           LIT        '1',
                   FALSE          LIT        '0',
                   FOREVER        LIT        'WHILE TRUE';



  4   1        DECLARE MAXRNO LITERALLY '104',/* MAX READ COUNT */
                       MAXLNO LITERALLY '129',/* MAX LOOK COUNT */
                       MAXPNO LITERALLY '145',/* MAX PUSH COUNT */
                       MAXSNO LITERALLY '234',/* MAX STATE COUNT */
                       STARTS LITERALLY '1',/* START STATE */

  5   1        DECLARE READ1 (*) BYTE
                   DATA(0, 57, 48, 56, 32, 8, 25, 59, 2, 16, 17, 22, 29, 53, 58, 11, 32, 32, 39
                   , 38, 34, 44, 9, 19, 32, 37, 6, 33, 3, 14, 15, 18, 20, 32, 28, 49, 32, 1, 42, 38, 36, 43, 1
                   , 1, 1, 1, 1, 1, 1, 1, 1, 10, 1, 39, 1, 1, 1, 38, 40, 49, 38, 39, 1, 1, 38, 23, 24, 55, 52, 41
                   , 35, 46, 1, 7, 50, 1, 32, 1, 32, 32, 45, 1, 32, 1, 32, 1, 32, 47, 37, 4, 26, 32, 54, 40, 1, 1
                   , 32, 5, 12, 13, 21, 22, 27, 1, 60, 1, 23, 24, 55, 30, 51);
  6   1        DECLARE LOOK1(*) BYTE
                   DATA(6, 8, 6, 25, 0, 9, 19, 0, 42, 0, 42, 0, 1, 0, 52, 0, 41, 0, 35, 0, 1, 0, 47
                   , 0, 4, 0, 54, 0, 40, 0, 35, 46, 60, 0, 1, 0, 32, 0, 1, 0, 1, 0, 11, 0, 60, 0, 7, 0, 32, 0, 32, 0
                   , 32, 0);
  7   1        DECLARE APPLY1(*) BYTE
                   DATA(0, 0, 0, 0, 0, 0, 9, 10, 12, 14, 19, 0, 0, 0, 0, 0, 0, 101, 0, 0, 100, 0
                   , 0, 0, 0, 0, 0, 97, 0, 27, 0, 0, 0, 63, 0, 91, 92, 0, 0, 91, 92, 0, 0, 0, 0, 13, 17, 0, 102
                   , 103, 104, 0, 0, 0, 0, 0, 95, 0, 0, 54, 0, 0, 23, 30, 38, 39, 0, 21, 46, 52, 56, 87, 93, 94
                   , 0);
  8   1        DECLARE READ2(*) BYTE
                   DATA(0, 65, 57, 64, 154, 26, 37, 67, 21, 30, 31, 33, 39, 61, 66, 27, 234
                   , 215, 51, 45, 108, 109, 213, 224, 233, 43, 216, 217, 22, 230, 229, 232, 231, 228, 173
                   , 172, 169, 9, 226, 47, 196, 195, 7, 8, 11, 13, 15, 2, 3, 105, 14, 158, 4, 50, 20, 12, 18
                   , 48, 171, 170, 44, 49, 19, 10, 46, 35, 36, 63, 60, 53, 42, 146, 16, 25, 58, 106, 155
                   , 148, 155, 155, 55, 150, 155, 152, 155, 157, 155, 56, 193, 23, 208, 234, 62, 52, 206
                   , 180, 234, 24, 28, 107, 32, 34, 38, 17, 68, 164, 35, 36, 63, 40, 59),
  9   1        DECLARE LOOK2(*) BYTE
                   DATA(0, 5, 130, 6, 131, 29, 29, 132, 41, 133, 54, 134, 135, 69, 71, 136
                   , 72, 137, 73, 138, 139, 80, 84, 140, 86, 138, 88, 141, 89, 142, 184, 184, 184, 91, 189
                   , 92, 93, 197, 211, 95, 143, 96, 97, 176, 39, 144, 145, 101, 102, 200, 103, 202, 104
                   , 188),
 10   1        DECLARE APPLY2(*) BYTE
                   DATA(0, 0, 77, 111, 112, 147, 79, 114, 81, 82, 83, 78, 76, 117, 75, 156
                   , 126, 163, 162, 100, 166, 165, 167, 118, 168, 160, 124, 179, 173, 94, 121, 74, 125
                   , 120, 119, 187, 187, 186, 98, 192, 192, 191, 194, 113, 183, 128, 129, 127, 205, 205
                   , 205, 204, 115, 123, 90, 122, 214, 213, 221, 219, 218, 222, 199, 85, 220, 116, 87
                   , 110, 70, 174, 209, 207, 182, 182, 181);
 11   1        DECLARE INDEX1(*) BYTE
                   DATA(0, 1, 2, 3, 4, 5, 6, 7, 8, 4, 4, 24, 4, 24, 4, 13, 14, 24, 109, 4, 15, 16
                   , 16, 24, 17, 18, 13, 16, 20, 22, 24, 25, 26, 28, 29, 34, 36, 37, 24, 24, 16, 38, 39, 40
                   , 42, 43, 44, 45, 46, 47, 48, 49, 16, 50, 38, 51, 16, 52, 53, 54, 55, 56, 57, 58, 60, 61
                   , 62, 63, 64, 8, 65, 68, 69, 70, 71, 72, 73, 74, 75, 77, 79, 81, 83, 85, 87, 88, 89, 90, 92
                   , 93, 94, 8, 8, 16, 95, 97, 97, 15, 103, 104, 105, 109, 24, 24, 24, 1, 3, 5, 8, 10, 12, 14
                   , 16, 18, 20, 22, 24, 26, 28, 30, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 185, 149, 225
                   , 227, 227, 190, 151, 153, 203, 159, 210, 161, 175, 212, 201, 177, 1, 2, 3, 3, 4, 4, 5, 5
                   , 6, 6, 12, 13, 14, 14, 15, 15, 16, 16, 17, 19, 19, 20, 20, 20, 22, 22, 23, 23, 24, 24, 25
                   , 25, 26, 26, 27, 29, 29, 31, 32, 32, 33, 33, 35, 38, 38, 33, 33, 39, 39, 39, 39, 39, 42
                   , 42, 43, 43, 44, 44, 45, 45, 48, 52, 52, 53, 53, 54, 54, 55, 55, 56, 56, 56, 56, 56, 56
                   , 56, 56, 58, 58, 58, 59, 59, 61, 61, 61, 61, 61, 62, 67),
 12   1        DECLARE INDEX2(*) BYTE
                   DATA(0, 1, 1, 1, 1, 1, 1, 1, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
```

```
             ,1,1,2,2,1,1,2,1,5,2,1,1,1,1,1,1,1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
             ,1,1,1,2,1,1,1,1,1,5,3,1,1,1,1,1,1,1,2,2,2,2,2,2,1,1,1,2,1,1,1,5,5,1
             ,2,6,6,1,1,1,4,2,1,1,1,2,2,3,2,2,2,2,2,2,2,2,2,2,2,4,2,2,2,2,2,2,2,2
             ,2,2,5,6,29,41,54,69,71,72,73,80,84,88,89,96,99,101,3,9,3,0,3,0,3,0
             ,0,1,7,8,1,0,6,0,0,1,3,0,1,1,2,1,0,0,0,0,1,0,2,0,0,1,2,0,1,5,3,0,0,1
             ,4,0,0,0,1,2,1,2,2,2,0,2,3,0,3,0,0,1,4,0,0,1,0,0,0,0,1,1,1,1,2,2,1,1
             ,1,0,0,0,0,0,0,0,0,0,0,0,0);

             /* END OF TABLES */
 1       DECLARE
             /* JOINT DECLARATIONS
             THESE ITEMS ARE DECLARED TOGETHER IN THIS SECTION
             IN ORDER TO FACILITATE THEIR BEING SAVED FOR
             THE SECOND PART OF THE COMPILER
             */

             OUTPUT$FCB      (33) BYTE INITIAL(0,'          ','CIN',0,0,0,0),
             DEBUGGING       BYTE      INITIAL (FALSE),
             PRINT$PROD      BYTE      INITIAL(FALSE),
             PRINT$TOKEN     BYTE      INITIAL(FALSE),
             LIST$INPUT      BYTE      INITIAL (TRUE),
             SEQ$NUM         BYTE      INITIAL (FALSE),
             NEXT$SYM        ADDRESS,
             POINTER         ADDRESS   INITIAL (100H),
             NEXT$AVAILABLE  ADDRESS   INITIAL (2002H),
             MAX$INT$MEM     ADDRESS   INITIAL (3200H),
             FILE$SEC$END    BYTE      INITIAL (FALSE),
             FREE$STORAGE    ADDRESS   INITIAL (2500H),

             /*  I O BUFFERS AND GLOBALS */
             IN$ADDR ADDRESS INITIAL (5CH),
             INPUT$FCB BASED INADDR (33) BYTE,
             OUTPUT$PTR  ADDRESS,
             OUTPUT$BUFF (128) BYTE,
             OUTPUT$END ADDRESS,
             OUTPUT$CHAR BASED OUTPUT$PTR BYTE;

 1       MON1: PROCEDURE (F,A) EXTERNAL;
 2           DECLARE A ADDRESS, F BYTE;
 2       END MON1;

 1       MON2: PROCEDURE (F,A) BYTE EXTERNAL;
 2           DECLARE F BYTE, A ADDRESS;
 2       END MON2;

 1       BOOT: PROCEDURE EXTERNAL;
 2           DECLARE A ADDRESS;
 2           END BOOT;


 1       PRINTCHAR: PROCEDURE (CHAR);
 2           DECLARE CHAR BYTE;
 2           CALL MON1 (2,CHAR);
 2       END PRINTCHAR;

 1       CRLF: PROCEDURE;
 2           CALL PRINTCHAR(CR);
 2           CALL PRINTCHAR(LF);
 2       END CRLF;

 1       PRINT: PROCEDURE (A);
 2           DECLARE A ADDRESS;
 2           CALL MON1 (9,A);
 2       END PRINT;

 1       PRINT$ERROR: PROCEDURE (CODE);
 2           DECLARE CODE ADDRESS;
 2           CALL CRLF;
 2           CALL PRINTCHAR(HIGH(CODE));
 2           CALL PRINTCHAR(LOW(CODE));
 2       END PRINT$ERROR;

 1       FATAL$ERROR: PROCEDURE(REASON);
 2           DECLARE REASON ADDRESS;
 2           CALL PRINT$ERROR(REASON);
 2           CALL TIME(10);
 2           CALL BOOT;
 2       END FATAL$ERROR;

 1       OPEN: PROCEDURE;
 2           IF MON2 (15,IN$ADDR)=255 THEN CALL FATAL$ERROR('OP');
 2       END OPEN;

 1       MORE$INPUT: PROCEDURE BYTE;
             /* READS THE INPUT FILE AND RETURNS TRUE IF A RECORD
                 WAS READ.  FALSE IMPLIES END OF FILE *
```

```
        DECLARE DCNT BYTE,
        IF (DCNT =MON2(20, INPUT$FCB))>1 THEN CALL FATAL$ERROR('BR'),
        RETURN NOT(DCNT),
    END MORE$INPUT,

    MAKE  PROCEDURE,
        /* DELETES ANY EXISTING COPY OF THE OUTPUT FILE
             AND CREATES A NEW COPY*/
        CALL MON1(19, OUTPUT$FCB),
        IF MON2(22, OUTPUT$FCB)=255 THEN CALL FATAL$ERROR('MA'),
    END MAKE,

    WRITE$OUTPUT  PROCEDURE,
        /* WRITES OUT A BUFFER */
        CALL MON1(26, OUTPUT$BUFF),      /* SET DMA */
        IF MON2(21, OUTPUT$FCB)<>0 THEN CALL FATAL$ERROR('WR'),
        CALL MON1(26,80H),      /* RESET DMA */
    END WRITE$OUTPUT,

    MOVE: PROCEDURE(SOURCE, DESTINATION, COUNT),
        /* MOVES FOR THE NUMBER OF BYTES SPECIFIED BY COUNT */
        DECLARE (SOURCE,DESTINATION) ADDRESS,
        (S$BYTE BASED SOURCE, D$BYTE BASED DESTINATION, COUNT) BYTE,
        DO WHILE (COUNT =COUNT - 1) <> 255,
             D$BYTE=S$BYTE,
             SOURCE=SOURCE +1,
             DESTINATION = DESTINATION + 1,
        END,
    END MOVE,

    FILL. PROCEDURE(ADDR,CHAR,COUNT),
        /* MOVES CHAR INTO ADDR FOR COUNT BYTES */
        DECLARE ADDR ADDRESS,
        (CHAR,COUNT,DEST BASED ADDR) BYTE,
        DO WHILE (COUNT =COUNT -1)<>255,
             DEST=CHAR,
             ADDR=ADDR + 1,
        END,
    END FILL,

        /*    *  *  *  *  *  SCANNER LITS *  *  *  *  */
    DECLARE
        LITERAL         LIT        '15',
        INPUT$STR       LIT        '32',
        PERIOD          LIT        '1',
        INVALID         LIT        '0';


        /*  * * * * SCANNER TABLES * * * *    */
    DECLARE TOKEN$TABLE (*) BYTE DATA
        /* CONTAINS THE TOKEN NUMBER ONE LESS THAN THE FIRST RESERVED WORD
        FOR EACH LENGTH OF WORD   */
        (0,0,1,4,5,15,22,32,38,44,47,49,51,55,56,57),

    TABLE (*) BYTE DATA('FD', OF','TO','PIC','COMP','DATA','FILE'
        ,'LEFT','MODE','SAME','SIGN','SYNC','ZERO','BLOCK','LABEL'
        ,'QUOTE','RIGHT','SPACE','USAGE','VALUE','ACCESS','ASSIGN'
        ,'AUTHOR','FILLER','OCCURS','RANDOM','RECORD', SELECT'
        ,'DISPLAY','LEADING , LINKAGE', OMITTED', RECORDS'
        ,'SECTION','DIVISION','RELATIVE','SECURITY','SEPARATE','STANDARD'
        ,'TRAILING','DEBUGGING','PROCEDURE','REDEFINES'
        ,'PROGRAM-ID','SEQUENTIAL','ENVIRONMENT','I-O-CONTROL
        ,'DATE-WRITTEN','FILE-CONTROL ,'INPUT-OUTPUT ,'ORGANIZATION'
        ,'CONFIGURATION','IDENTIFICATION','OBJECT-COMPUTER'
        ,'SOURCE-COMPUTER','WORKING-STORAGE'),

    OFFSET  (16) ADDRESS
        /*  NUMBER OF BYTES TO INDEX INTO THE TABLE FOR EACH LENGTH */
        INITIAL (0,0,0,6,9,45,80,123,170,218,245,265,
             287,325,348,362),

    WORD$COUNT (*) BYTE DATA
        /* NUMBER OF WORDS OF EACH SIZE */
        (0,0,3,1,9,7,8,6,6,3,2,2,4,1,1,3),


        MAX$LEN         LIT        '16',
        ADD$END(*) BYTE DATA        ('PROCEDURE '),
        LOOKED          BYTE       INITIAL (0),
        HOLD            BYTE,
        BUFFER$END      ADDRESS    INITIAL    (100H),
        NEXT            BASED       POINTER  BYTE,
        INBUFF          LIT        '80H',
        CHAR            BYTE,
        ACCUM$LENG      LIT        '50',
        ACCUM$LEN$P$1   LIT        '51',    /*  = TO ACCUM$LENG PLUS 1  */
        ACCUM (ACCUM$LEN$P$1) BYTE,
```

```
    DISPLAY(74)     BYTE        INITIAL (0),
    TOKEN           BYTE;             /*RETURNED FROM SCANNER */

    /* * * * *   PROCEDURES USED BY THE SCANNER * * * */

NEXT$CHAR:  PROCEDURE BYTE;
    IF LOOKED THEN
    DO;
         LOOKED=FALSE;
         RETURN (CHAR =HOLD);
    END;
    IF (POINTER:=POINTER + 1) >= BUFFER$END THEN
    DO;
        IF NOT MORE$INPUT THEN
         DO;
              BUFFER$END= MEMORY;
              POINTER= ADD$END;
         END;
         ELSE POINTER=INBUFF;
    END;
    RETURN (CHAR:=NEXT);
END NEXT$CHAR;

GET$CHAR:  PROCEDURE;
    /* THIS PROCEDURE IS CALLED WHEN A NEW CHAR IS NEEDED WITHOUT
    THE DIRECT RETURN OF THE CHARACTER*/
    CHAR=NEXT$CHAR;
END GET$CHAR;

DISPLAY$LINE: PROCEDURE;
    IF NOT LIST$INPUT THEN RETURN;
    DISPLAY(DISPLAY(0) + 1) = '$';
    CALL PRINT( DISPLAY(1));
    DISPLAY(0) = 0;
END DISPLAY$LINE;


LOAD$DISPLAY: PROCEDURE;
    IF DISPLAY(0) < 72 THEN
         DISPLAY(DISPLAY(0) =DISPLAY(0) + 1) = CHAR;
    CALL GET$CHAR;
END LOAD$DISPLAY;

PUT:  PROCEDURE;
    IF ACCUM(0) < ACCUM$LENG THEN
    ACCUM(ACCUM(0) =ACCUM(0)+1)=CHAR;
    CALL LOAD$DISPLAY;
END PUT;

EAT$LINE: PROCEDURE;
    DO WHILE CHAR<>CR;
         CALL LOAD$DISPLAY;
    END;
END EAT$LINE;

GET$NO$BLANK: PROCEDURE;
    DECLARE (N, I) BYTE;
    DO FOREVER;
         IF CHAR = ' ' THEN CALL LOAD$DISPLAY;
         ELSE
         IF CHAR=CR THEN
         DO;
              CALL DISPLAY$LINE;
              IF SEQ$NUM THEN N=8; ELSE N=2;
              DO I = 1 TO N;
                   CALL LOAD$DISPLAY;
              END;
              IF CHAR = '*' THEN CALL EAT$LINE;
              ELSE
              IF CHAR = '.' THEN
                   DO;
                        IF NOT DEBUGGING THEN CALL EAT$LINE;
                        ELSE CALL LOAD$DISPLAY;
                   END;
         END;
         ELSE
         RETURN;
    END;    /* END OF DO FOREVER */
END GET$NO$BLANK;

SPACE: PROCEDURE BYTE;
    RETURN (CHAR= ' ') OR (CHAR=CR);
END SPACE;

DELIMITER:  PROCEDURE BYTE;
    /* CHECKS FOR A PERIOD FOLLOWED BY A SPACE OR CR*/
```

```
2            IF CHAR <> ' ' THEN RETURN FALSE;
2            HOLD=NEXT$CHAR;
2            LOOKED=TRUE;
2            IF SPACE THEN
2            DO;
3                 CHAR = ' ';
3                 RETURN TRUE;
3            END;
2            CHAR=' ';
2            RETURN FALSE;
2        END DELIMITER;

1        END$OF$TOKEN: PROCEDURE BYTE;
2            RETURN SPACE OR  DELIMITER;
2        END END$OF$TOKEN;

1        GET$LITERAL: PROCEDURE BYTE;
2            CALL LOAD$DISPLAY;
2            DO FOREVER;
3                 IF CHAR= QUOTE THEN
3                 DO;
4                      CALL LOAD$DISPLAY;
4                      RETURN LITERAL;
4                 END;
3                 CALL PUT;
3            END;
2        END GET$LITERAL;


1        LOOK$UP: PROCEDURE BYTE;
2            DECLARE POINT ADDRESS,
             HERE BASED POINT (1) BYTE,
             I                    BYTE;

2            MATCH: PROCEDURE BYTE;
3                 DECLARE J BYTE;
3                 DO J=1 TO ACCUM(0);
4                      IF HERE(J - 1) <> ACCUM(J) THEN RETURN FALSE;
4                 END;
3                 RETURN TRUE;
3            END MATCH;

2            POINT=OFFSET(ACCUM(0))+ TABLE;
2            DO I=1 TO WORD$COUNT(ACCUM(0));
3                 IF MATCH THEN RETURN I;
3                 POINT = POINT + ACCUM(0);
3            END;
2            RETURN FALSE;
2        END LOOK$UP;

1        RESERVED$WORD: PROCEDURE BYTE;
             /* RETURNS THE TOKEN NUMBER OF A RESERVED WORD IF THE CONTENTS OF
             THE ACCUMULATOR IS A RESERVED WORD, OTHERWISE RETURNS ZERO */
2            DECLARE VALUE BYTE;
2            DECLARE NUMB BYTE;
2                 IF ACCUM(0) > MAX$LEN THEN RETURN 0;
2                 IF (NUMB:=TOKEN$TABLE(ACCUM(0)))=0 THEN RETURN 0;
2                 IF (VALUE =LOOK$UP)=0 THEN RETURN 0;
2                 RETURN (NUMB + VALUE);
2        END RESERVED$WORD;

1        GET$TOKEN: PROCEDURE BYTE;
2            ACCUM(0)=0;
2            CALL GET$NO$BLANK;
2            IF CHAR=QUOTE THEN RETURN GET$LITERAL;
2            IF DELIMITER THEN
2            DO;
3                 CALL PUT;
3                 RETURN PERIOD;
3            END;
2            DO FOREVER;
3                 CALL PUT;
3                 IF END$OF$TOKEN THEN RETURN INPUT$STR;
3            END; /* OF DO FOREVER */
2        END GET$TOKEN;


1        SCANNER: PROCEDURE;
2            DECLARE CHECK BYTE;
2            DO FOREVER;
3                 IF(TOKEN =GET$TOKEN) = INPUT$STR THEN
3                      IF (CHECK:=RESERVED$WORD) <> 0 THEN TOKEN=CHECK;
3                 IF TOKEN <> 0 THEN RETURN;
3                 CALL PRINT$ERROR ('SE');
3                 DO WHILE NOT END$OF$TOKEN;
4                      CALL GET$CHAR;
```

```
                END.
            END.
        END SCANNER;


        PRINT$ACCUM. PROCEDURE;
            ACCUM(ACCUM(0)+1)='$',
            CALL PRINT( ACCUM(1));
        END PRINT$ACCUM;

        PRINT$NUMBER: PROCEDURE(NUMB),
            DECLARE(NUMB, I, CNT, K) BYTE,  J(*) BYTE DATA(100, 10);
            DO I=0 TO 1,
                CNT=0;
                DO WHILE NUMB >= (K.=J(I));
                    NUMB=NUMB - K;
                    CNT=CNT + 1,
                END;
                CALL PRINTCHAR('0' + CNT),
            END;
            CALL PRINTCHAR('0' + NUMB),
        END PRINT$NUMBER,


        INIT$SCANNER: PROCEDURE,
            DECLARE CON$CBL (*) BYTE DATA ('CBL');
            /*    INITIALIZE FOR INPUT - OUTPUT OPERATIONS     */
            CALL MOVE ( CON$CBL, IN$ADDR + 9, 3),
            CALL FILL(IN$ADDR + 12, 0, 5);
            CALL OPEN;
            CALL MOVE(INADDR, OUTPUT$FCB, 9),
            OUTPUT$FCB(32) = 0,
            OUTPUT$END=(OUTPUT$PTR.=.OUTPUT$BUFF - 1) + 128;
            CALL MAKE;
            CALL GET$CHAR,     /* PRIME THE SCANNER */
            DO WHILE CHAR = '$',
                IF NEXTCHAR = 'L' THEN LIST$INPUT=NOT LIST$INPUT;
                ELSE IF CHAR ='S' THEN SEQ$NUM= NOT SEQ$NUM,
                ELSE IF CHAR = 'P' THEN PRINT$PROD = NOT PRINT$PROD,
                ELSE IF CHAR = 'T' THEN PRINT$TOKEN = NOT PRINT$TOKEN;
                CALL GET$CHAR;
                CALL GET$NO$BLANK;
            END;
        END INIT$SCANNER;

            /*  *   *   *   END OF SCANNER PROCEDURES   *   *   *   */


            /*  *   *   *   *   SYMBOL TABLE DECLARATIONS * * *   */

        DECLARE

        CUR$SYM              ADDRESS,         /*SYMBOL BEING ACCESSED*/
        SYMBOL               BASED CUR$SYM (1)  BYTE,
        SYMBOL$ADDR          BASED CUR$SYM (1) ADDRESS,
        NEXT$SYM$ENTRY       BASED NEXT$SYM ADDRESS,
        HASH$PTR             ADDRESS,
        DISPLACEMENT          LIT              '12',
        HASH$MASK            LIT              '3FH',
        S$TYPE               LIT              '2',
        OCCURS               LIT              '11',
        ADDR2                LIT              '4',
        P$LENGTH             LIT              '3',
        S$LENGTH             LIT              '3',
        LEVEL                LIT              '10',
        LOCATION             LIT              '2',
        REL$ID               LIT              '5',
        START$NAME           LIT              '11',    /*1 LESS*/
        MAX$ID$LEN           LIT              '12',

            /*  *   *   *   *   TYPE LITERALS * * * * * * * */

        DECLARE
        SEQUENTIAL     LIT        '1',
        RANDOM         LIT        '2',
        SEQ$RELATIVE   LIT        '3',
        VARIABLE$LENG  LIT        '4',
        GROUP          LIT        '6',
        COMP           LIT        '21',

            /*  *   *   *   SYMBOL TABLE ROUTINES   *   *   *   */

        INIT$SYMBOL  PROCEDURE,
            CALL FILL (FREE$STORAGE, 0, 130);
            /* INITIALIZE HASH TABLE AND FIRST COLLISION FIELD */
            NEXT$SYM=FREE$STORAGE+128;
```

```
2            NEXT$SYM$ENTRY=0;
2       END INIT$SYMBOL;

1       GET$P$LENGTH  PROCEDURE BYTE;
2            RETURN SYMBOL(P$LENGTH);
2       END GET$P$LENGTH;

1       SET$ADDRESS  PROCEDURE(ADDR);
2       DECLARE ADDR ADDRESS;
2            SYMBOL$ADDR(LOCATION)=ADDR;
2       END SET$ADDRESS;

1       GET$ADDRESS  PROCEDURE ADDRESS;
2            RETURN SYMBOL$ADDR(LOCATION);
2       END GET$ADDRESS;

1       GET$TYPE  RROCEDURE BYTE;
2            RETURN SYMBOL(S$TYPE);
2       END GET$TYPE;

1       SET$TYPE  PROCEDURE(TYPE);
2            DECLARE TYPE BYTE;
2            SYMBOL(S$TYPE)=TYPE;
2       END SET$TYPE;

1       OR$TYPE  PROCEDURE(TYPE);
2            DECLARE TYPE BYTE;
2            SYMBOL(S$TYPE)=TYPE OR GET$TYPE;
2       END OR$TYPE;

1       GET$LEVEL  PROCEDURE BYTE;
2            RETURN SHR(SYMBOL(LEVEL), 4);
2       END GET$LEVEL;

1       SET$LEVEL  PROCEDURE (LVL);
2            DECLARE LVL BYTE;
2            SYMBOL(LEVEL)=SHL(LVL, 4) OR SYMBOL(LEVEL);
2       END SET$LEVEL;

1       GET$DECIMAL: PROCEDURE BYTE;
2            RETURN SYMBOL(LEVEL) AND 0FH;
2       END GET$DECIMAL;

1       SET$DECIMAL  PROCEDURE (DEC);
2            DECLARE DEC BYTE;
2            SYMBOL(LEVEL) = DEC OR SYMBOL(LEVEL);
2       END SET$DECIMAL;

1       SET$S$LENGTH  PROCEDURE(HOW$LONG);
2            DECLARE HOW$LONG ADDRESS;
2            SYMBOL$ADDR(S$LENGTH) = HOW$LONG;
2       END SET$S$LENGTH;

1       GET$S$LENGTH  PROCEDURE ADDRESS;
2            RETURN SYMBOL$ADDR(S$LENGTH);
2       END GET$S$LENGTH;


1       SET$ADDR2: PROCEDURE (ADDR);
2            DECLARE ADDR ADDRESS;
2       SYMBOL$ADDR(ADDR2)=ADDR;
2       END SET$ADDR2;

1       GET$ADDR2  PROCEDURE ADDRESS;
2            RETURN SYMBOL$ADDR(ADDR2);
2       END GET$ADDR2;

1       SET$OCCURS  PROCEDURE(OCCUR);
2            DECLARE OCCUR BYTE;
2            SYMBOL(OCCURS)=OCCUR;
2       END SET$OCCURS;

1       GET$OCCURS  PROCEDURE BYTE;
2            RETURN SYMBOL (OCCURS);
2       END GET$OCCURS;

             /*  *  *  *  PARSER DECLARATIONS  *  *  *  */
1       DECLARE
        INT                   LIT        '63',       /* CODE FOR INITIALIZE */
        SCD                   LIT        '66',       /* CODE FOR SET CODE START */
        PSTACKSIZE             LIT        '30',       /* SIZE OF PARSE STACKS*/
        STATESTACK            (PSTACKSIZE)    BYTE,   /* SAVED STATES */
        VALUE                 (PSTACKSIZE)    ADDRESS,     /* TEMP VALUES */
        VARC                  (51)        BYTE,       /*TEMP CHAR STORE*/
        ID$STACK              (10)      ADDRESS     INITIAL (0),
        ID$STACK$PTR          BYTE      INITIAL(0),
        HOLD$LIT (ACCUM$LEN$P$1)    BYTE,
```

```
HOLD$SYM              ADDRESS,
PENDING$LITERAL       BYTE INITIAL(FALSE),
PENDING$LIT$ID        ADDRESS,
REDEF                 BYTE      INITIAL (FALSE),
REDEF$ONE             ADDRESS,
REDEF$TWO             ADDRESS,
TEMP$HOLD             ADDRESS,
TEMP$TWO              ADDRESS,
COMPILING             BYTE   INITIAL(TRUE),
SP                    BYTE   INITIAL (255),
MP                    BYTE,
MPP1                  BYTE,
NOLOOK                BYTE   INITIAL(TRUE),
(I, J, K)             BYTE,         /*INDICIES FOR THE PARSER*/
STATE                 BYTE   INITIAL(STARTS);

    /*  *  *  *  PARSER ROUTINES  *  *  *  *  */

BYTE$OUT  PROCEDURE(ONE$BYTE);
    /* THIS PROCEDURE WRITES ONE BYTE OF OUTPUT ONTO THE DISK
    IF REQUIRED THE OUTPUT BUFFER IS DUMPED TO THE DISK */
    DECLARE ONE$BYTE BYTE;
    IF (OUTPUT$PTR:=OUTPUT$PTR + 1)) OUTPUT$END THEN
    DO;
         CALL WRITE$OUTPUT;
         OUTPUT$PTR= OUTPUT$BUFF;
    END;
    OUTPUT$CHAR=ONE$BYTE;
END BYTE$OUT;

STRING$OUT  PROCEDURE (ADDR, COUNT);
    DECLARE (ADDR, I, COUNT) ADDRESS, (CHAR BASED ADDR) BYTE;
    DO I=1 TO COUNT;
         CALL BYTE$OUT(CHAR);
         ADDR=ADDR+1;
    END;
END STRING$OUT;

ADDR$OUT  PROCEDURE(ADDR);
    DECLARE ADDR ADDRESS;
    CALL BYTE$OUT(LOW(ADDR));
    CALL BYTE$OUT(HIGH(ADDR));
END ADDR$OUT;

FILL$STRING: PROCEDURE(COUNT, CHAR);
    DECLARE (I, COUNT) ADDRESS, CHAR BYTE;
    DO I=1 TO COUNT;
         CALL BYTE$OUT(CHAR);
    END;
END FILL$STRING;

START$INITIALIZE: PROCEDURE(ADDR, CNT);
    DECLARE (ADDR, CNT) ADDRESS;
    CALL BYTEOUT(INT);
    CALL ADDR$OUT(ADDR);
    CALL ADDR$OUT(CNT);
END START$INITIALIZE;

BUILD$SYMBOL  PROCEDURE(LEN);
    DECLARE LEN BYTE, TEMP ADDRESS;
    TEMP=NEXT$SYM;
    IF (NEXT$SYM = SYMBOL(LEN:=LEN+DISPLACEMENT))
         > MAX$MEMORY THEN CALL FATAL$ERROR('ST');
    CALL FILL (TEMP, 0, LEN);
END BUILD$SYMBOL;

MATCH: PROCEDURE ADDRESS;
    /* CHECKS AN IDENTIFIER TO SEE IF IT IS IN THE SYMBOL
    TABLE.  IF IT IS PRESENT, CUR$SYM IS SET FOR ACCESS
    OTHERWISE A NEW ENTRY IS MADE AND THE PRINT NAME
    IS ENTERED.  ALL NAMES ARE TRUNCATED TO MAX$ID$LEN*/
    DECLARE POINT                   ADDRESS,
            COLLISION BASED POINT ADDRESS,
            (HOLD, I)               BYTE;
    IF VARC(0)>MAX$ID$LEN
         THEN VARC(0) = MAX$ID$LEN;
         /* TRUNCATE IF REQUIRED */
    HOLD = 0;
    DO I=1 TO VARC(0);      /* CALCULATE HASH CODE */
         HOLD=HOLD + VARC(I);
    END;
    POINT=FREE$STORAGE + SHL((HOLD AND HASH$MASK), 1);
    DO FOREVER;
        IF COLLISION=0 THEN
         DO;
              CUR$SYM, COLLISION=NEXT$SYM;
              CALL BUILD$SYMBOL(VARC(0));
```

```
                    /* LOAD PRINT NAME */
                    SYMBOL(P$LENGTH)=VARC(0);
                    DO I = 1 TO VARC(0);
                         SYMBOL(START$NAME + I)=VARC(I);
                    END;
                    RETURN CUR$SYM;
               END;
               ELSE
               DO;
                    CUR$SYM=COLLISION;
                    IF (HOLD =GET$P$LENGTH)=VARC(0) THEN
                    DO;
                        I=1;
                        DO WHILE SYMBOL(START$NAME + I)= VARC(I);
                         IF (I =I+1)>HOLD THEN RETURN (CUR$SYM =COLLISION);
                        END;
                    END;
               END;
               POINT=COLLISION;
          END;
     END MATCH;

     ALLOCATE: PROCEDURE(BYTES$REQ) ADDRESS;
          /* THIS ROUTINE CONTROLS THE ALLOCATION OF SPACE
          IN THE MEMORY OF THE INTERPRETER    */

          DECLARE (HOLD,BYTES$REQ) ADDRESS;
          HOLD=NEXT$AVAILABLE;
          IF (NEXT$AVAILABLE =NEXT$AVAILABLE + BYTES$REQ)>MAX$INT$MEM
               THEN CALL FATAL$ERROR('MO');
          RETURN HOLD;
     END ALLOCATE;

     SET$REDEF: PROCEDURE(OLD,NEW);
          DECLARE (OLD,NEW) ADDRESS;
          IF (REDEF =NOT REDEF) THEN
          DO;
               REDEF$ONE=OLD;
               REDEF$TWO=NEW;
          END;
          ELSE CALL PRINT$ERROR('R1');
     END SET$REDEF;

     SET$CUR$SYM: PROCEDURE;
          CUR$SYM=ID$STACK(ID$STACK$PTR);
     END SET$CUR$SYM;

     STACK$LEVEL: PROCEDURE BYTE;
          CALL SET$CUR$SYM;
          RETURN GET$LEVEL;
     END STACK$LEVEL;

     LOAD$LEVEL: PROCEDURE;
          DECLARE HOLD ADDRESS;

          LOAD$REDEF$ADDR: PROCEDURE;
               CUR$SYM=REDEF$ONE;
               HOLD=GET$ADDRESS;
          END LOAD$REDEF$ADDR;

          IF ID$STACK(0) <> 0 THEN
          DO;
               IF VALUE(SP-2)=0 THEN
               DO;
                    CALL SET$CUR$SYM;
                    HOLD=GET$S$LENGTH + GET$ADDRESS;
               END;
               ELSE CALL LOAD$REDEF$ADDR;
               IF (ID$STACK$PTR =ID$STACK$PTR+1)>9 THEN
               DO;
                    CALL PRINT$ERROR('EL');
                    ID$STACK$PTR=9;
               END;
          END;
          ELSE HOLD=NEXT$AVAILABLE;
          ID$STACK(ID$STACK$PTR)=VALUE(MPP1);
          CALL SET$CUR$SYM;
          CALL SET$ADDRESS(HOLD);
     END LOAD$LEVEL;

     REDEF$OR$VALUE: PROCEDURE;
          DECLARE HOLD ADDRESS,
               (DEC,K,J,SIGN) BYTE;
          IF REDEF THEN
          DO;
               IF REDEF$TWO=CUR$SYM THEN
               DO;
```

```
                    HOLD=GET$S$LENGTH;
                    CUR$SYM=REDEF$ONE;
                    IF HOLD>GET$S$LENGTH THEN
                    DO;
                         CALL PRINT$ERROR('R2');
                         HOLD=GET$S$LENGTH;
                         CUR$SYM=REDEF$ONE;
                         CALL SET$S$LENGTH(HOLD);
                    END;
                    REDEF=FALSE;
               END;
          END;
          ELSE IF PENDING$LITERAL=0 THEN RETURN;
          IF PENDING$LIT$ID<>ID$STACK$PTR THEN RETURN;
          CALL START$INITIALIZE(GET$ADDRESS, HOLD =GET$S$LENGTH);
          IF PENDING$LITERAL>2 THEN
          DO;
               IF PENDING$LITERAL=3 THEN CHAR='0';
               ELSE IF PENDING$LITERAL=4 THEN CHAR=' ';
               ELSE CHAR=QUOTE;
               CALL FILL$STRING(HOLD, CHAR);
          END;
          ELSE IF PENDING$LITERAL = 2 THEN
          DO;
               IF HOLD <= HOLD$LIT(0) THEN
                    CALL STRING$OUT( HOLD$LIT(1), HOLD);
               ELSE DO;
                    CALL STRING$OUT( HOLD$LIT(1), HOLD$LIT(0));
                    CALL FILL$STRING(HOLD - (HOLD$LIT(0) + 1), ' ');
               END;
          END;
          ELSE DO;
               /* THE NUMBER HANDELER */
               DECLARE (DEC, MINUS$SIGN, I, J, LIT$DEC, N$LENGTH,
                    NUM$BEFORE, NUM$AFTER, TYPE) BYTE, ZONE LIT '10H';

               IF((TYPE =GET$TYPE)<16) OR (TYPE>20) THEN
                    CALL PRINT$ERROR('NV');
               N$LENGTH=GET$S$LENGTH;
               DEC=GET$DECIMAL;
               MINUS$SIGN=FALSE;
               IF HOLD$LIT(1) = '-' THEN
               DO;
                    MINUS$SIGN=TRUE;
                    J=1;
               END;
               ELSE IF HOLD$LIT(1) = '+' THEN J=1;
               ELSE J=0;
               LIT$DEC=0;
               DO I=1 TO HOLD$LIT(0);
                    IF HOLD$LIT(I)='.' THEN LIT$DEC=I;
               END;
               IF LIT$DEC=0 THEN
               DO;
                    NUM$BEFORE=HOLD$LIT(1)-J;
                    NUM$AFTER=0;
               END;
               ELSE DO;
                    NUM$BEFORE=LIT$DEC -J-1;
                    NUM$AFTER=HOLD$LIT(1) - LIT$DEC;
               END;
               IF (I =N$LENGTH - DEC)<NUM$BEFORE THEN
                    CALL PRINT$ERROR('SL');
                    IF I>NUM$BEFORE THEN
                    DO;
                    I=I-NUM$BEFORE;
                    IF MINUS$SIGN THEN
                    DO;
                         I=I-1;
                         CALL BYTE$OUT('0' + ZONE);
                    END;
                    CALL FILL$STRING(I,'0');
               END;
               ELSE IF MINUS$SIGN THEN HOLD$LIT(J+1)=HOLD$LIT(J+1)+ZONE;
               CALL STRING$OUT( HOLD$LIT(1) + J, NUM$BEFORE);
               IF NUM$AFTER > DEC THEN NUM$AFTER = DEC;
               CALL STRING$OUT( HOLD$LIT(1) + LIT$DEC, NUM$AFTER);
               IF (I =DEC - NUM$AFTER)<>0 THEN
                    CALL FILL$STRING(I,'0');
          END;
          PENDING$LITERAL=0;
     END REDEF$OR$VALUE;

     REDUCE$STACK  PROCEDURE;
          DECLARE HOLD$LENGTH ADDRESS;
          CALL SET$CUR$SYM;
          CALL REDEF$OR$VALUE
```

```
        HOLD$LENGTH=GET$S$LENGTH;
        IF GET$TYPE > 128 THEN
        DO;
            HOLD$LENGTH=HOLD$LENGTH + GET$OCCURS;
        END;
        ID$STACK$PTR=ID$STACK$PTR - 1;
        CALL SET$CUR$SYM;
        CALL SET$S$LENGTH(GET$S$LENGTH + HOLD$LENGTH);
        CALL SET$TYPE(GROUP);
    END REDUCE$STACK;

    END$OF$RECORD: PROCEDURE;
        DO WHILE ID$STACK$PTR<>0;
            CALL REDUCE$STACK;
        END;
        CALL SET$CUR$SYM;
        CALL REDEF$OR$VALUE;
        ID$STACK(0)=0;
        TEMP$HOLD=ALLOCATE(TEMP$TWO =GET$S$LENGTH);
    END  END$OF$RECORD;

    CONVERT$INTEGER:  PROCEDURE;
        DECLARE INTEGER ADDRESS;
        INTEGER=0;
        DO I = 1 TO VARC(0);
            INTEGER=SHL(INTEGER,3)+SHL(INTEGER,1)+(VARC(I) - '0');
        END;
        VALUE(SP)=INTEGER;
    END CONVERT$INTEGER;

    OR$VALUE: PROCEDURE(PTR,ATTRIB);
        DECLARE PTR BYTE, ATTRIB ADDRESS;
        VALUE(PTR)=VALUE(PTR) OR ATTRIB;
    END OR$VALUE;

    BUILD$FCB:  PROCEDURE;
        DECLARE TEMP ADDRESS;
        DECLARE BUFFER(11) BYTE, (CHAR, I, J) BYTE;
        CALL FILL(.BUFFER,' ',11);
        J,I=0;
        DO WHILE (J < 11) AND (I< VARC(0));
            IF (CHAR =VARC(I =I+1))=' ' THEN J=8;
            ELSE DO;
                BUFFER(J)=CHAR;
                J=J+1;
            END;
        END;
        CALL SET$ADDR2(TEMP =ALLOCATE(164));
        CALL START$INITIALIZE(TEMP,16 );
        CALL BYTE$OUT(0);
        CALL STRING$OUT(.BUFFER,11);
        CALL FILL$STRING(4,0);
        CALL OR$VALUE(SP-1,1);
    END BUILD$FCB;

    SET$SIGN:  PROCEDURE(NUMB);
        DECLARE NUMB BYTE;
        IF GET$TYPE=17 THEN CALL SET$TYPE(VALUE(SP) + NUMB);
        ELSE CALL PRINT$ERROR('SG');
        IF VALUE(SP)<>0 THEN CALL SET$S$LENGTH(GET$S$LENGTH + 1);
    END SET$SIGN;

    PIC$ANALIZER:  PROCEDURE;
        DECLARE    /* WORK AREAS AND VARIABLES */
        FLAG          BYTE,
        FIRST         BYTE,
        COUNT         ADDRESS,
        BUFFER (31)   BYTE,
        SAVE          BYTE,
        REPITITIONS   ADDRESS,
        J             BYTE,
        DEC$COUNT     BYTE,
        CHAR          BYTE,
        I             BYTE,
        TEMP          ADDRESS,
        TYPE          BYTE,

         /* * * MASKS * * */
        ALPHA     LIT  '0',
        A$EDIT    LIT  '2',
        A$N       LIT  '4',
        EDIT      LIT  '8',
        NUM       LIT  '16',
        NUM$EDIT  LIT  '32',
        DEC       LIT  '64',
        SIGN      LIT  '128',
```

```
NUM$MASK        LIT          10101111B',
NUM$ED$MASK     LIT          10000101B',
S$NUM$MASK      LIT         '00101111B',
A$E$MASK        LIT          11111100B ,
A$N$MASK        LIT          11101010B',
A$N$E$MASK      LIT         '11100000B ,

/* TYPES */
NETYPE LIT '80',
NTYPE  LIT  16 ,
SNTYPE LIT '17',
ATYPE  LIT '8',
AETYPE LIT '72',
ANTYPE LIT '9',
ANETYPE LIT '73',

INC$COUNT  PROCEDURE(SWITCH),
     DECLARE SWITCH BYTE,
     FLAG=FLAG OR SWITCH;
     IF (COUNT =COUNT + 1) < 31 THEN BUFFER(COUNT) = CHAR,
END INC$COUNT,

CHECK  PROCEDURE (MASK) BYTE,
     /* THIS ROUTINE CHECKS A MASK AGINST THE
     FLAG BYTE AND RETURNS TRUE ID THE FLAG
     HAD NO BITS IN COMMON WITH THE MASK */
     DECLARE MASK BYTE,
     RETURN NOT ( (FLAG AND MASK) <> 0),
END CHECK,

PIC$ALLOCATE  PROCEDURE(AMT) ADDRESS,
     DECLARE AMT ADDRESS,
     IF (MAX$INT$MEM:=MAX$INT$MEM - AMT) < NEXT$AVAILABLE
          THEN CALL FATAL$ERROR ('NO'),
     RETURN MAX$INT$MEM,
END PIC$ALLOCATE.

/* RROCEDURE EXECUTION STARTS HERE */

COUNT, FLAG, DEC$COUNT=0;
/* CHECK FOR EXCESSIVE LENGTH */
IF VARC(0) > 30 THEN
DO;
     CALL PRINT$ERROR('PC'),
     RETURN,
END;
/* SET FLAG BITS AND COUNT LENGTH */
I =1;
DO WHILE I<=VARC(0);
     IF (CHAR =VARC(I))='A' THEN CALL INC$COUNT(ALPHA),
     ELSE IF CHAR ='B' THEN CALL INC$COUNT(A$EDIT),
     ELSE IF CHAR ='9' THEN CALL INC$COUNT(NUM);
     ELSE IF CHAR ='X' THEN CALL INC$COUNT(A$N),
     ELSE IF (CHAR='S') AND (COUNT=0) THEN
          FLAG=FLAG OR SIGN;
     ELSE IF (CHAR = 'V') AND (DEC$COUNT=0) THEN
          DEC$COUNT=COUNT,
     ELSE IF(CHAR='/') OR (CHAR='0') THEN CALL INC$COUNT(EDIT),
     ELSE IF
          (CHAR='Z') OR (CHAR=',') OR (CHAR='*') OR
          (CHAR='+') OR (CHAR='-') OR (CHAR='$') THEN
          CALL INC$COUNT(NUM$EDIT),
     ELSE IF (CHAR=' ') AND (DEC$COUNT=0) THEN
     DO,
          CALL INC$COUNT(NUM$EDIT);
          DEC$COUNT=COUNT,
     END,
     ELSE IF ((CHAR='C') AND (VARC(I+1)='R')) OR
          ((CHAR='D') AND (VARC(I+1)='B')) THEN
     DO;
          CALL INC$COUNT(NUM$EDIT),
          CHAR=VARC(I =I+1),
          CALL INC$COUNT(NUM$EDIT),
     END,
     ELSE IF (CHAR='(') AND (COUNT<>0) THEN
     DO,
          SAVE=VARC(I-1),
          REPITITIONS=0;
          DO WHILE(CHAR =VARC(I =I+1))<>')',
               REPITITIONS=SHL(REPITITIONS,3) +
               SHL(REPITITIONS,1) +(CHAR -'0');
          END,
          CHAR=SAVE,
          DO J=1 TO REPITITIONS-1,
               CALL INC$COUNT(0),
          END,
     END,
```

```
                 ELSE DO;
                      CALL PRINT$ERROR('PC');
                      RETURN;
                 END;
                 I=I+1;
            END; /* END OF DO WHILE I<= VARC */
            /* AT THIS POINT THE TYPE CAN BE DETERMINED */
            IF  NOT CHECK(NUM$EDIT) THEN
            DO;
                 IF CHECK(NUM$ED$MASK) THEN TYPE=NETYPE;
            END;
            ELSE IF CHECK(NUM$MASK) THEN TYPE=NTYPE;
            ELSE IF CHECK(SNUM$MASK) THEN TYPE=S$N$TYPE;
            ELSE IF CHECK(NOT(ALPHA)) THEN TYPE=ATYPE;
            ELSE IF CHECK(A$E$MASK) THEN TYPE =AETYPE;
            ELSE IF CHECK(A$N$MASK) THEN TYPE=ANTYPE;
            ELSE IF CHECK(A$N$E$MASK) THEN TYPE=ANETYPE;
            IF TYPE=0 THEN CALL PRINT$ERROR('PC');
            ELSE DO;
                 IF REDEF THEN CUR$SYM=REDEF$TWO;
                 ELSE CUR$SYM = HOLD$SYM;
                 CALL SET$TYPE(TYPE);
                 CALL SET$SLENGTH(COUNT + GET$S$LENGTH);
                 IF (TYPE AND 64) <> 0 THEN
                 DO;
                      CALL SET$ADDR2(TEMP :=PIC$ALLOCATE(COUNT));
                      CALL START$INITIALIZE(TEMP, COUNT);
                      CALL STRING$OUT(.BUFFER + 1, COUNT);
                 END;
                 IF DEC$COUNT<>0 THEN CALL SET$DECIMAL(COUNT-DEC$COUNT);
            END;
        END PIC$ANALIZER;

        SET$FILE$ATTRIB: PROCEDURE;
            DECLARE TEMP ADDRESS, TYPE BYTE;
            IF CUR$SYM<>VALUE(MPP1) THEN
            DO;
                 TEMP=CUR$SYM;
                 CUR$SYM=VALUE(MPP1);
                 SYMBOL$ADDR(REL$ID)=TEMP;
            END;
            IF NOT (TEMP:=VALUE(SP-1)) THEN CALL PRINT$ERROR ('NF');
            ELSE  DO;
                 IF TEMP=1 THEN TYPE=SEQUENTIAL;
                 ELSE IF TEMP=15 THEN TYPE=RANDOM;
                 ELSE IF TEMP=9 THEN TYPE=SEQ$RELATIVE;
                 ELSE DO;
                      CALL PRINT$ERROR('IA');
                           TYPE=1;
                 END;
                 END;
            CALL SET$TYPE(TYPE);
        END SET$FILE$ATTRIB;

        LOAD$LITERAL: PROCEDURE;
            DECLARE I BYTE;
            IF PENDING$LITERAL <> 0 THEN CALL PRINT$ERROR ('LE');
            ELSE DO I = 0 TO VARC(0);
                 HOLD$LIT(I)=VARC(I);
            END;
        END LOAD$LITERAL;


        CHECK$FOR$LEVEL: PROCEDURE;
              DECLARE NEW$LEVEL BYTE;
              HOLD$SYM, CUR$SYM=VALUE(MP-1);
              CALL SET$LEVEL(NEW$LEVEL:=VALUE(MP-2));
              IF NEW$LEVEL=1 THEN
              DO;
                   IF ID$STACK(0)<>0 THEN
                   DO;
                        IF NOT FILE$SEC$END THEN
                        DO;
                             CALL SET$REDEF(ID$STACK(0),VALUE(MP-1));
                             VALUE(MP)=1;     /* SET REDEFINE FLAG */
                        END;
                        CALL END$OF$RECORD;
                   END;
              END;
              ELSE DO WHILE STACK$LEVEL >= NEW$LEVEL;
                   CALL REDUCE$STACK;
              END;
        END CHECK$FOR$LEVEL;


        CODE$GEN: PROCEDURE(PRODUCTION);
             DECLARE PRODUCTION BYTE;
```

```
        IF PRINT$PROD THEN
        DO;
             CALL CRLF;
             CALL PRINTCHAR(POUND);
             CALL PRINT$NUMBER(PRODUCTION);
        END;

        DO CASE PRODUCTION;

      /*  P R O D U C T I O N S  */

    /* CASE 0 NOT USED                                                 */
              ;
    /*        1   <PROGRAM>    = <ID-DIV> <E-DIV> <D-DIV> PROCEDURE      */
        COMPILING=FALSE;
    /*        2   <ID-DIV>    = IDENTIFICATION DIVISION   PROGRAM-ID     */
    /*        2                <COMMENT>   <AUTH> <DATE> <SEC>           */
        ;    /* NO ACTION REQUIRED */
    /*        3   <AUTH>     = AUTHOR    <COMMENT>                        */
        ;    /* NO ACTION REQUIRED */
    /*        4               \! <EMPTY>                                   */
        ;    /* NO ACTION REQUIRED */
    /*        5   <DATE>     = DATE-WRITTEN   <COMMENT>                   */
        ;    /* NO ACTION REQUIRED */
    /*        6               \! <EMPTY>                                   */
        ;    /* NO ACTION REQUIRED */
    /*        7   <SEC>     = SECURITY    <COMMENT>                       */
        ;    /* NO ACTION REQUIRED */
    /*        8              \! <EMPTY>                                    */
        ;    /* NO ACTION REQUIRED */
    /*        9   <COMMENT>   ::= <INPUT>                                 */
        ;    /* NO ACTION REQUIRED */
    /*       10               \! <COMMENT> <INPUT>                         */
        ;    /* NO ACTION REQUIRED */
    /*       11   <E-DIV>    = ENVIRONMENT DIVISION   CONFIGURATION       */
    /*       11              SECTION    <SRC-OBJ> <I-O>                   */
        ;    /* NO ACTION REQUIRED */
    /*       12   <SRC-OBJ>    = SOURCE-COMPUTER   <COMMENT> <DEBUG>      */
    /*       12                  OBJECT-COMPUTER   <COMMENT>              */
        ;    /* NO ACTION REQUIRED */
    /*       13   <DEBUG>    = DEBUGGING MODE                             */
        DEBUGGING=TRUE;   /* SETS A SCANNER TOGGLE */
    /*       14              \! <EMPTY>                                    */
        ;    /* NO ACTION REQUIRED */
    /*       15   <I-O> ::= INPUT-OUTPUT SECTION   FILE-CONTROL           */
    /*       15              <FILE-CONTROL-LIST> <IC>                     */
        ;    /* NO ACTION REQUIRED */
    /*       16              \! <EMPTY>                                    */
        ;    /* NO ACTION REQUIRED */
    /*       17   <FILE-CONTROL-LIST>   := <FILE-CONTROL-ENTRY>           */
        ;    /* NO ACTION REQUIRED */
    /*       18                        \! <FILE-CONTROL-LIST>              */
    /*       18                          <FILE-CONTROL-ENTRY>             */
        ;    /* NO ACTION REQUIRED */
    /*       19   <FILE-CONTROL-ENTRY>   = SELECT <ID> <ATTRIBUTE-LIST>   */
        CALL SET$FILE$ATTRIB;
    /*       20   <ATTRIBUTE-LIST>    = <ONE-ATTRIB>                      */
        ;    /* NO ACTION REQUIRED */
    /*       21                      \! <ATTRIBUTE-LIST> <ONE-ATTRIB>      */
        VALUE(MP)=VALUE(SP) OR VALUE(MP);
    /*       22   <ONE-ATTRIB>    := ORGANIZATION <ORG-TYPE>              */
        VALUE(MP)=VALUE(SP);
    /*       23                  \! ACCESS <ACC-TYPE> <RELATIVE>           */
        VALUE(MP)=VALUE(MPP1) OR VALUE(SP);
    /*       24                  \! ASSIGN <INPUT>                         */
        CALL BUILD$FCB;
    /*       25   <ORG-TYPE>    = SEQUENTIAL                              */
        ;    /* NO ACTION REQUIRED - DEFAULT */
    /*       26               \! RELATIVE                                  */
        CALL OR$VALUE(SP,4);
    /*       27   <ACC-TYPE>    = SEQUENTIAL                              */
        ;    /* NO ACTION REQUIRED - DEFAULT */
    /*       28               \! RANDOM                                    */
        CALL OR$VALUE(SP,2);
    /*       29   <RELATIVE>    = RELATIVE <ID>                           */
        CALL OR$VALUE(MP,8);
    /*       30               \! <EMPTY>                                   */
        ;    /* NO ACTION REQUIRED - DEFAULT */
    /*       31   <IC>     = I-O-CONTROL    <SAME-LIST>                   */
        ;
    /*       32            \! <EMPTY>                                      */
        ;
    /*       33   <SAME-LIST>    = <SAME-ELEMENT>                         */
        ;
    /*       34                \! <SAME-LIST> <SAME-ELEMENT>               */
        ;
    /*       35   <SAME-ELEMENT>    = SAME <ID-STRING>                    */
```

```
         ;
/*       36   <ID-STRING>    = <ID>                                                  */
         ,
/*       37                    \! <ID-STRING> <ID>                                    */
         ;

/*       38   <D-DIV>    = DATA DIVISION    <FILE-SECTION> <WORK>                     */
/*       38                   <LINK>                                                 */
         ;    /* NO ACTION REQUIRED */
/*       39   <FILE-SECTION>    = FILE SECTION    <FILE-LIST>                         */
         FILE$SEC$END = TRUE;
/*       40                       \! <EMPTY>                                          */
         FILE$SEC$END=TRUE;
/*       41   <FILE-LIST>    = <FILES>                                               */
         ,    /* NO ACTION REQUIRED */
/*       42                    \! <FILE-LIST> <FILES>                                 */
         ,    /* NO ACTION REQUIRED */
/*       43   <FILES>    = FD <ID> <FILE-CONTROL>                                     */
/*       43                   <RECORD-DESCRIPTION>                                   */
         DO;
              CALL END$OF$RECORD;
              CUR$SYM=VALUE(MPP1);
              CALL SET$ADDRESS(TEMP$HOLD);
              CALL SET$S$LENGTH(TEMP$TWO);
         END;
/*       44   <FILE-CONTROL>    = <FILE-LIST>                                         */
         ,    /* NO ACTION REQUIRED */
/*       45                       \! <EMPTY>                                          */
         ,    /* NO ACTION REQUIRED */
/*       46   <FILE-LIST>    = <FILE-ELEMENT>                                         */
         ,    /* NO ACTION REQUIRED */
/*       47                    \! <FILE-LIST> <FILE-ELEMENT>                          */
         ;    /* NO ACTION REQUIRED */
/*       48   <FILE-ELEMENT>    = BLOCK <INTEGER> RECORDS                             */
         ;    /* NO ACTION REQUIRED - FILES NEVER BLOCKED   */
/*       49                       \! RECORD <REC-COUNT>                               */
         CALL SET$SLENGTH(VALUE(SP));
/*       50                       \! LABEL RECORDS STANDARD                           */
         ,    /* NO ACTION REQUIRED */
/*       51                       \! LABEL RECORDS OMITTED                            */
         .    /* NO ACTION REQUIRED */
/*       52                       \! VALUE OF <ID-STRING>                             */
         ,    /* NO ACTION REQUIRED */
/*       53   <REC-COUNT>    = <INTEGER>                                              */
         .    /* NO ACTION REQUIRED - VALUE(SP) CORRECT */
/*       54                    \! <INTEGER> TO <INTEGER>                              */
         DO;
         VALUE(MP)=VALUE(SP);  /* VARIABLE LENGTH */
         CALL SET$TYPE(4);     /* SET TO VARIABLE */
         END;
/*       55   <WORK>    = WORKING-STORAGE SECTION                                     */
/*       55                <RECORD-DESCRIPTION>                                       */
         ,    /* NO ACTION REQUIRED */
/*       56                \! <EMPTY>                                                 */
         ,    /* NO ACTION REQUIRED */
/*       57   <LINK>    = LINKAGE SECTION    <RECORD-DESCRIPTION>                     */
         CALL PRINT$ERROR('NI'); /* INTER PROG COMM */
/*       58                \! <EMPTY>                                                 */
         ,    /* NO ACTION REQUIRED */
/*       59   <RECORD-DESCRIPTION>    = <LEVEL-ENTRY>                                 */
         .    /* NO ACTION REQUIRED */
/*       60                                 \! <RECORD-DESCRIPTION>                    */
/*       60                                    <LEVEL-ENTRY>                           */
         ,    /* NO ACTION REQUIRED */
/*       61   <LEVEL-ENTRY>    = <INTEGER> <DATA-ID> <REDEFINES>                      */
/*       61                      <DATA-TYPE>                                          */
         DO;
              CALL LOAD$LEVEL;
              IF PENDING$LITERAL<>0 THEN PENDING$LIT$ID=ID$STACK$PTR;
         END;
/*       62   <DATA-ID>    = <ID>                                                    */
         ,    /* NO ACTION REQUIRED */
/*       63                   \! FILLER                                              */
         DO;
              CUR$SYM, VALUE(SP)=NEXT$SYM;
              CALL BUILD$SYMBOL(0);
         END;
/*       64   <REDEFINES>    = REDEFINES <ID>                                         */
         DO;
              CALL SET$REDEF(VALUE(SP), VALUE(SP-2));
              VALUE(MP)=1;      /* SET REDEFINE FLAG ON */
              CALL CHECK$FOR$LEVEL;
         END;
/*       65                      \! <EMPTY>                                           */
         CALL CHECK$FOR$LEVEL;
/*       66   <DATA-TYPE>    = <PROP-LIST>                                            */
         ,     /* NO ACTION REQUIRED */
```

```
    /*      67                   \! <EMPTY>                                          */
        ;     /* NO ACTION REQUIRED */
    /*      68    <PROP-LIST>    = <DATA-ELEMENT>                                   */
        ;     /* NO ACTION REQUIRED */
    /*      69                   \! <PROP-LIST> <DATA-ELEMENT>                       */
        ;     /* NO ACTION REQUIRED */
    /*      70    <DATA-ELEMENT>    = PIC <INPUT>                                   */
       CALL PIC$ANALIZER;
    /*      71                   \! USAGE COMP                                       */
       CALL SET$TYPE(COMP);
    /*      72                   \! USAGE DISPLAY                                    */
        ;     /* NO ACTION REQUIRED - DEFAULT */
    /*      73                   \! SIGN LEADING <SEPARATE>                          */
       CALL SET$SIGN(18);
    /*      74                   \! SIGN TRAILING <SEPARATE>                         */
       CALL SET$SIGN(17);
    /*      75                   \! OCCURS <INTEGER>                                 */
       DO;
            CALL OR$TYPE(128);
            CALL SET$OCCURS(VALUE(SP));
       END;
    /*      76                   \! SYNC <DIRECTION>                                 */
        ;     /* NO ACTION REQUIRED - BYTE MACHINE */
    /*      77                   \! VALUE <LITERAL>                                  */
       DO;
            IF NOT FILE$SEC$END THEN
            DO;
                 CALL PRINT$ERROR('VE');
                 PENDING$LITERAL=0;
            END;
       END;
    /*      78    <DIRECTION>    = LEFT                                             */
        ;     /* NO ACTION REQUIRED */
    /*      79                   \! RIGHT                                            */
        ;     /* NO ACTION REQUIRED */
    /*      80                   \! <EMPTY>                              */
        ;     /* NO ACTION REQUIRED */
    /*      81    <SEPARATE>     = SEPARATE                                          */
       VALUE(SP)=2;
    /*      82                   \! <EMPTY>                                          */
        ;     /* NO ACTION REQUIRED */
    /*      83    <LITERAL>      = <INPUT>                                          */
       DO;
            CALL LOAD$LITERAL;
            PENDING$LITERAL=1;
       END;
    /*      84                   \! <LIT>                                            */
       DO;
            CALL LOAD$LITERAL;
            PENDING$LITERAL=2;
       END;
    /*      85                   \! ZERO                                             */
       PENDING$LITERAL=3;
    /*      86                   \! SPACE                                            */
       PENDING$LITERAL=4;
    /*      87                   \! QUOTE                                            */
       PENDING$LITERAL=5;
    /*      88    <INTEGER>      = <INPUT>                                          */
       CALL CONVERT$INTEGER;
    /*      89    <ID>           = <INPUT>                                          */
       VALUE(SP)=MATCH;     /* STORE SYMBOL TABLE POINTERS */


       END;    /* END OF CASE STATEMENT */
   END CODE$GEN;

   GETIN1  PROCEDURE BYTE;
       RETURN INDEX1(STATE);
   END GETIN1;

   GETIN2  PROCEDURE BYTE;
       RETURN INDEX2(STATE);
   END GETIN2;

   INCSP  PROCEDURE;
       SP=SP + 1;
       IF SP >= PSTACKSIZE THEN CALL FATAL$ERROR('SO');
       VALUE(SP)=0;     /* CLEAR VALUE STACK */
   END INCSP;

   LOOKAHEAD  PROCEDURE;
       IF NOLOOK THEN
       DO;
            CALL SCANNER;
            NOLOOK=FALSE;
            IF PRINT$TOKEN THEN
                 DO;
```

```
4                  CALL CRLF;
4                  CALL PRINT$NUMBER(TOKEN);
4                  CALL PRINT$CHAR(' ');
4                  CALL PRINT$ACCUM;
4              END;
3          END;
2      END LOOKAHEAD;

1      NO$CONFLICT  PROCEDURE (CSTATE) BYTE;
2          DECLARE (CSTATE, I, J, K) BYTE;
2          J=INDEX1(CSTATE);
2          K=J + INDEX2(CSTATE) - 1;
2          DO I=J TO K;
3              IF READ1(I)=TOKEN THEN RETURN TRUE;
3          END;
2      RETURN FALSE;
2      END NO$CONFLICT;

1      RECOVER  PROCEDURE BYTE;
2          DECLARE (TSP, RSTATE) BYTE;
2          DO FOREVER;
3              TSP=SP;
3              DO WHILE TSP <> 255;
4                  IF NO$CONFLICT(RSTATE:=STATESTACK(TSP)) THEN
4                  DO;   /* STATE WILL READ TOKEN */
5                      IF SP<>TSP THEN SP = TSP - 1;
5                      RETURN RSTATE;
5                  END;
4                  TSP = TSP - 1;
4              END;
3              CALL SCANNER;   /* TRY ANOTHER TOKEN */
3          END;
2      END RECOVER;

1      END$PASS  PROCEDURE;
           /* THIS PROCEDURE STORES THE INFORMATION REQUIRED BY PASS2
           IN LOCATIONS ABOVE THE SYMBOL TABLE. THE FOLLOWING
           INFORMATION IS STORED
               OUTPUT FILE CONTROL BLOCK
               COMPILER TOGGLES
               INPUT BUFFER POINTER
           THE OUTPUT BUFFER IS ALSO FILLED SO THE CURRENT RECORD IS WRITTEN.
           */

2          CALL BYTE$OUT(SCO);
2          CALL ADDR$OUT(NEXT$AVAILABLE);
2          DO WHILE OUTPUT$PTR<> OUTPUT$BUFF;
3              CALL BYTE$OUT(0FFH);
3          END;

2          CALL MOVE( OUTPUT$FCB, MAX$MEMORY-PASS1$LEN, PASS1$LEN);
2          L   GO TO L;     /*   PATCH TO "JMP   3100H"   */
2      END END$PASS;

           /*  *  *  *  *  PROGRAM EXECUTION STARTS HERE *  *  */

1      CALL MOVE(INITIAL$POS, MAX$MEMORY, RDR$LENGTH);
1      CALL INIT$SCANNER;
1      CALL INIT$SYMBOL;

           /*  *  *  *  *  *  *  PARSER *  *  *  *  *  *  */

1      DO WHILE COMPILING;
2          IF STATE <= MAXRNO THEN          /* READ STATE */
2          DO;
3              CALL INCSP;
3              STATESTACK(SP) = STATE;   /* SAVE CURRENT STATE */
3              CALL LOOKAHEAD;
3              I=GETIN1;
3              J = I + GETIN2 - 1;
3              DO I=I TO J;
4                  IF READ1(I) = TOKEN THEN
4                  DO;
                   /* COPY THE ACCUMULATOR IF IT IS AN INPUT
                   STRING. IF IT IS A RESERVED WORD IT DOES
                   NOT NEED TO BE COPIED  */
5                      IF (TOKEN=INPUT$STR) OR (TOKEN=LITERAL) THEN
5                          DO K=0 TO ACCUM(0);
6                              VARC(K)=ACCUM(K);
6                          END;
5                      STATE=READ2(I);
5                      NOLOOK=TRUE;
5                      I=J;
5                  END;
                   ELSE
4                  IF I=J THEN
```

```
                        DO;
                             CALL PRINT$ERROR('NP');
                             CALL PRINT(.(' ERROR NEAR $'));
                             CALL PRINT$ACCUM;
                             IF (STATE =RECOVER)=0 THEN COMPILING=FALSE;
                        END;
                   END;
              END;      /* END OF READ STATE */
              ELSE
              IF STATE>MAXPNO THEN       /* APPLY PRODUCTION STATE */
              DO;
                   MP=SP - GETIN2;
                   MPP1=MP + 1;
                   CALL CODE$GEN(STATE - MAXPNO);
                   SP=MP;
                   I=GETIN1;
                   J=STATESTACK(SP);
                   DO WHILE (K =APPLY1(I)) <> 0 AND J<>K;
                        I=I + 1;
                   END;
                   IF (K =APPLY2(I))=0 THEN COMPILING=FALSE;
                   STATE=K;
              END;
              ELSE
              IF STATE<=MAXLNO THEN     /*LOOKAHEAD STATE*/
              DO;
              I=GETIN1;
                   CALL LOOKAHEAD;
                   DO WHILE (K =LOOK1(I))<>0 AND TOKEN <>K;
                        I=I+1;
                   END;
              STATE=LOOK2(I);
              END;
              ELSE
              DO;          /*PUSH STATES*/
                   CALL INCSP;
                   STATESTACK(SP)=GETIN2;
                   STATE=GETIN1;
              END;
         END;   /* OF WHILE COMPILING */
         CALL CRLF;
         CALL PRINT( .('PROCEDURES'));
         CALL END$PASS;
         END;


MODULE INFORMATION

     CODE AREA SIZE     = 1E91H   7825D
     VARIABLE AREA SIZE = 02FCH    764D
     MAXIMUM STACK SIZE = 001CH     28D
     1517 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE INTERP
OBJECT MODULE PLACED IN  F1 INTERP OBJ
COMPILER INVOKED BY   PLM80  F1 INTERP PLM


                $PAGELENGTH(90)
  1             INTERP:      /*  MODULE  " I N T E R P "     */
                       DO;

                    /*          COBOL INTERPRETER    _ */

                 /*       NORMALLY ORG'ED TO X'100'       */

                    /* GLOBAL DECLARATIONS AND LITERALS    */

  2    1        DECLARE

                LIT       LITERALLY       'LITERALLY',
                BDOS           LIT        '5H',      /* ENTRY TO OPERATING SYSTEM */
                BOOT           LIT        '0',
                CR             LIT        '13',
                LF             LIT        '10',
                TRUE           LIT        '1',
                FALSE          LIT        '0',
                FOREVER        LIT        'WHILE TRUE',

                    /* UTILITY VARIABLES */

  3    1        DECLARE

                BOOTER         ADDRESS                  INITIAL (0000H),
                INDEX          BYTE,
                A$CTR          ADDRESS,
                CTR            BYTE,
                BASE           ADDRESS,
                B$BYTE         BASED BASE (1)           BYTE,
                B$ADDR         BASED BASE (1)           ADDRESS,
                HOLD           ADDRESS,
                H$BYTE         BASED HOLD (1)           BYTE,
                H$ADDR         BASED HOLD (1)           ADDRESS,

                    /* CODE POINTERS */

                CODE$START          LIT       '2000H',
                PROGRAM$COUNTER     ADDRESS,
                C$BYTE              BASED PROGRAM$COUNTER (1)   BYTE,
                C$ADDR              BASED PROGRAM$COUNTER (1)   ADDRESS,

                    /* * * * *   GLOBAL INPUT AND OUTPUT ROUTINES * * * * */


  4    1        DECLARE
                CURRENT$FCB ADDRESS,
                START$OFFSET        LIT       '36',

  5    1        MON1: PROCEDURE (F,A) EXTERNAL;
  6    2            DECLARE F BYTE, A ADDRESS;
  7    2        END MON1;

  8    1        MON2: PROCEDURE (F,A) BYTE EXTERNAL;
  9    2            DECLARE F BYTE, A ADDRESS;
 10    2        END MON2;

 11    1        PRINT$CHAR: PROCEDURE (CHAR);
 12    2            DECLARE CHAR BYTE;
 13    2            CALL MON1 (2,CHAR);
 14    2        END PRINT$CHAR;

 15    1        CRLF: PROCEDURE;
 16    2            CALL PRINT$CHAR(CR);
 17    2            CALL PRINT$CHAR(LF);
 18    2        END CRLF;

 19    1        PRINT: PROCEDURE (A);
 20    2            DECLARE A ADDRESS;
 21    2            CALL CRLF;
 22    2            CALL MON1(9,A);
 23    2        END PRINT;


 24    1        READ: PROCEDURE(A);
 25    2            DECLARE A ADDRESS;
 26    2            CALL MON1(10,A);
 27    2        END READ;
```

```
PRINT$ERROR: PROCEDURE (CODE);
    DECLARE CODE ADDRESS;
    CALL CRLF;
    CALL PRINT$CHAR(HIGH(CODE));
    CALL PRINT$CHAR(LOW(CODE));
END PRINT$ERROR;

FATAL$ERROR: PROCEDURE(CODE);
    DECLARE CODE ADDRESS;
    CALL PRINT$ERROR(CODE);
    CALL BOOTER;
END FATAL$ERROR;

SET$DMA: PROCEDURE;
    CALL MON1 (26, CURRENT$FCB + START$OFFSET);
END SET$DMA;

OPEN: PROCEDURE (ADDR) BYTE;
    DECLARE ADDR ADDRESS;
    CALL SET$DMA;    /* INSURE DIRECTORY READ WON'T CLOBBER CORE */
    RETURN MON2(15,ADDR);
END OPEN;

CLOSE: PROCEDURE (ADDR);
    DECLARE ADDR ADDRESS;
    IF MON2(16,ADDR)=255 THEN CALL FATAL$ERROR('CL');
END CLOSE;

DELETE: PROCEDURE;
    CALL MON1(19,CURRENT$FCB);
END DELETE;

MAKE: PROCEDURE (ADDR);
    DECLARE ADDR ADDRESS;
    IF MON2(22,ADDR)=255 THEN CALL FATAL$ERROR('ME');
END MAKE;

DISK$READ: PROCEDURE BYTE;
    RETURN MON2(20,CURRENT$FCB);
END DISK$READ;

DISK$WRITE: PROCEDURE BYTE;
    RETURN MON2(21,CURRENT$FCB);
END DISK$WRITE;

    /* * * * * * * * * * UTILITY PROCEDURES * * * * * * * * * * * * * */

DECLARE
SUBSCRIPT          (8)        ADDRESS;

RES: PROCEDURE(ADDR) ADDRESS;
    /* THIS PROCEDURE RESOLVES THE ADDRESS OF A SUBSCRIPTED
    IDENTIFIER OR A LITERAL CONSTANT */

    DECLARE ADDR ADDRESS;
    IF ADDR > 32 THEN RETURN ADDR;
    IF ADDR < 9 THEN RETURN SUBSCRIPT(ADDR);
    DO CASE ADDR - 9;
        RETURN  ('0');
        RETURN  (' ');
        RETURN  ('"');
    END;
    RETURN 0;
END RES;

MOVE: PROCEDURE(FROM,DESTINATION,COUNT);
    DECLARE (FROM,DESTINATION,COUNT) ADDRESS,
        (F BASED FROM, D BASED DESTINATION) BYTE;
    DO WHILE (COUNT:=COUNT - 1) <> 0FFFFH;
        D=F;
        FROM=FROM + 1;
        DESTINATION=DESTINATION + 1;
    END;
```

```
END MOVE;

FILL: PROCEDURE(DESTINATION,COUNT,CHAR);
    DECLARE (DESTINATION,COUNT) ADDRESS,

        (CHAR,D BASED DESTINATION) BYTE;
    DO WHILE (COUNT =COUNT - 1)<> 0FFFFH;
        D=CHAR;
        DESTINATION=DESTINATION + 1;
    END;
END FILL;

CONVERT$TO$HEX: PROCEDURE(POINTER,COUNT) ADDRESS;
    DECLARE POINTER ADDRESS, COUNT BYTE;
    A$CTR=0;
    BASE=POINTER;
    DO CTR = 0 TO COUNT-1;
        A$CTR=SHL(A$CTR,3) + SHL(A$CTR,1) + B$BYTE(CTR) - '0';
    END;
    RETURN A$CTR;
END CONVERT$TO$HEX;

    /* * * * * * * * * * * CODE CONTROL PROCEDURES * * * * * * * * * */

DECLARE

BRANCH$FLAG          BYTE        INITIAL(FALSE);

INC$PTR: PROCEDURE (COUNT);
    DECLARE COUNT BYTE;
    PROGRAM$COUNTER=PROGRAM$COUNTER + COUNT;
END INC$PTR;

GET$OP$CODE: PROCEDURE BYTE;
    CTR=C$BYTE(0);
    CALL INC$PTR(1);
    RETURN CTR;
END GET$OP$CODE;

COND$BRANCH: PROCEDURE(COUNT);
    /* THIS PROCEDURE CONTROLS BRANCHING INSTRUCTIONS */
    DECLARE COUNT BYTE;
    IF  BRANCH$FLAG THEN
    DO;
        BRANCH$FLAG=FALSE;
        PROGRAM$COUNTER=C$ADDR(COUNT);
    END;
    ELSE CALL INC$PTR(SHL(COUNT,1)+2);
END COND$BRANCH;

INCR$OR$BRANCH: PROCEDURE(MARK);
    DECLARE MARK BYTE;
    IF MARK THEN CALL INC$PTR(2);
    ELSE PROGRAM$COUNTER=C$ADDR(0);
END INCR$OR$BRANCH;

    /* * * * * * * * * * * *COMPARISONS * * * * * * * * * * * * * * * */


CHAR$COMPARE: PROCEDURE BYTE;
    BASE=C$ADDR(0);
    HOLD=C$ADDR(1);
    DO A$CTR=0 TO C$ADDR(2) - 1;
        IF B$BYTE(A$CTR) > H$BYTE(A$CTR) THEN RETURN 1;
        IF B$BYTE(A$CTR) < H$BYTE(A$CTR) THEN RETURN 0;
    END;
    RETURN 2;
END CHAR$COMPARE;

STRING$COMPARE: PROCEDURE(PIVOT);
    DECLARE PIVOT BYTE;
    IF CHAR$COMPARE=PIVOT THEN BRANCH$FLAG=NOT BRANCH$FLAG;
    CALL COND$BRANCH(3);
END STRING$COMPARE;

NUMERIC: PROCEDURE(CHAR) BYTE;
    DECLARE CHAR BYTE;
```

```
  2           RETURN (CHAR >='0') AND (CHAR <='9');
  2       END NUMERIC;

  1       LETTER: PROCEDURE(CHAR) BYTE;
  2           DECLARE CHAR BYTE;
  2           RETURN (CHAR >='A') AND (CHAR <='Z');
  2       END LETTER;

  1       SIGN: PROCEDURE(CHAR) BYTE;
  2           DECLARE CHAR BYTE;
  2           RETURN (CHAR='+') OR (CHAR='-');
  2       END SIGN;

  1       COMP$NUM$UNSIGNED: PROCEDURE;
  2           BASE=C$ADDR(0);
  2           DO A$CTR=0 TO C$ADDR(2)-1;
  3               IF NOT NUMERIC(B$BYTE(A$CTR)) THEN
  3               DO;
  4                   BRANCH$FLAG=NOT BRANCH$FLAG;
  4                   RETURN;
  4               END;
  3           END;
  2           CALL COND$BRANCH(2);
  2       END COMP$NUM$UNSIGNED;

  1       COMP$NUM$SIGN: PROCEDURE;
  2           BASE=C$ADDR(0);
  2           DO A$CTR=0 TO C$ADDR(2)-1;
  3               IF NOT(NUMERIC(CTR:=B$BYTE(A$CTR))
                      OR SIGN(CTR)) THEN
  3               DO;
  4                   BRANCH$FLAG=NOT BRANCH$FLAG;
  4                   RETURN;
  4               END;
  3           END;
  2           CALL COND$BRANCH(2);
  2       END COMP$NUM$SIGN;

  1       COMP$ALPHA: PROCEDURE;
  2           BASE=C$ADDR(0);
  2           DO A$CTR=0 TO C$ADDR(2)-1;
  3               IF NOT LETTER(B$BYTE(A$CTR)) THEN
  3               DO;
  4                   BRANCH$FLAG=NOT BRANCH$FLAG;
  4                   RETURN;
  4               END;
  3           END;
  2           CALL COND$BRANCH(2);
  2       END COMP$ALPHA;


            /* * * * * * * * * * *NUMERIC OPERATIONS * * * * * * * * * * * */


  1       DECLARE

          (R0,R1,R2)          (10)      BYTE, /* REGISTERS */
          SIGN0(3)           BYTE,
          (DEC$PT0,DEC$PT1,DEC$PT2)     BYTE,
          DEC$PTA (3)          BYTE AT ( DEC$PT0),
          OVERFLOW             BYTE,
          R$PTR                BYTE,
          SWITCH               BYTE,
          SIGNIF$NO            BYTE,
          ZONE                 LIT       '10H',
          POSITIVE             LIT       '1',
          NEGITIVE             LIT       '0';

  1       CHECK$FOR$SIGN: PROCEDURE(CHAR) BYTE;
  2           DECLARE CHAR BYTE;
  2           IF NUMERIC(CHAR) THEN RETURN POSITIVE;
  2           IF NUMERIC(CHAR - ZONE) THEN RETURN NEGITIVE;
  2           CALL PRINT$ERROR('SI');
  2           RETURN POSITIVE;
  2       END CHECK$FOR$SIGN;

  1       STORE$IMMEDIATE: PROCEDURE;
  2           DO CTR=0 TO 9;
  3               R0(CTR)=R2(CTR);
```

```
  3             END;
  2             DEC$PT0=DEC$PT2;
  2             SIGN0(0)=SIGN0(2);
  2         END STORE$IMMEDIATE;


  1         ONE$LEFT: PROCEDURE;
  2             DECLARE (CTR, FLAG) BYTE;
  2             IF ((FLAG =SHR(B$BYTE(0),4))=0) OR (FLAG=9) THEN
  2             DO;
  3                 DO CTR=0 TO 8;
  4                     B$BYTE(CTR)=SHL(B$BYTE(CTR),4) OR SHR(B$BYTE(CTR + 1),4);
  4                 END;
  3                 B$BYTE(9)=SHL(B$BYTE(9),4) OR FLAG;
  3             END;
  2             ELSE OVERFLOW=TRUE;
  2         END ONE$LEFT;


  1         ONE$RIGHT: PROCEDURE;
  2             DECLARE CTR BYTE;
  2             CTR=10;
  2             DO INDEX=1 TO 9;
  3                 CTR=CTR-1;
  3                 B$BYTE(CTR)=SHR(B$BYTE(CTR),4) OR SHL(B$BYTE(CTR-1),4);
  3             END;
  2             B$BYTE(0)=SHR(B$BYTE(0),4);
  2             IF B$BYTE(0) = 09H THEN
  2                 B$BYTE(0) = 99H;
  2         END ONE$RIGHT;


  1         SHIFT$RIGHT: PROCEDURE(COUNT);
  2             DECLARE COUNT BYTE;
  2             DO CTR=1 TO COUNT;
  3                 CALL ONE$RIGHT;
  3             END;
  2         END SHIFT$RIGHT;


  1         SHIFT$LEFT: PROCEDURE (COUNT);
  2             DECLARE COUNT BYTE;
  2             OVERFLOW=FALSE;
  2             DO CTR=1 TO COUNT;
  3                 CALL ONE$LEFT;
  3                 IF OVERFLOW THEN RETURN;
  3             END;
  2         END SHIFT$LEFT;


  1         ALLIGN: PROCEDURE;
  2             BASE= R0;
  2             IF DEC$PT0 > DEC$PT1 THEN CALL SHIFT$RIGHT(DEC$PT0-DEC$PT1);
  2             ELSE CALL SHIFT$LEFT(DEC$PT1-DEC$PT0);
  2         END ALLIGN;


  1         ADD$R0: PROCEDURE(SECOND, DEST);
  2             DECLARE (SECOND, DEST) ADDRESS, (CY, A, B, I, J) BYTE;
  2             HOLD= SECOND;
  2             BASE = DEST;
  2             CY=0;
  2             CTR=9;
  2             DO J=1 TO 10;
  3                 A=R0(CTR);
  3                 B=H$BYTE(CTR);
  3                 I=DEC(A+CY);
  3                 CY=CARRY;
  3                 I=DEC(I + B);
  3                 CY=(CY OR CARRY) AND 1;
  3                 B$BYTE(CTR)=I;
  3                 CTR=CTR-1;
  3             END;
  2             IF CY THEN
  2             DO;
  3                 CTR=9;
  3                 DO J = 1 TO 10;
  4                     I=B$BYTE(CTR);
  4                     I=DEC(I+CY);
  4                     CY=CARRY AND 1;
  4                     B$BYTE(CTR)=I;
  4                     CTR=CTR-1;
  4                 END;
  3             END;
  2         END ADD$R0;
```

```
COMPLIMENT  PROCEDURE(NUMB);
    DECLARE NUMB BYTE;

    SIGN0(NUMB) = SIGN0(NUMB) XOR 1;   /* COMPLIMENT SIGN */

    DO CASE NUMB;
         HOLD= R0;
         HOLD= R1;
         HOLD= R2;
    END;

    DO CTR=0 TO 9;
         H$BYTE(CTR)=99H - H$BYTE(CTR);
    END;

END COMRLIMENT;



R2$ZERO: PROCEDURE BYTE;
   DECLARE I BYTE;
   IF (SHL(R2(0),4)<>0) OR (SHR(R2(9),4)<>0)
   THEN RETURN FALSE;
   ELSE DO I=1 TO 8;
          IF R2(I)<>0 THEN RETURN FALSE;
        END;
   RETURN TRUE;
END R2$ZERO;

CHECK$RESULT  PROCEDURE;
    IF SHR(R2(0),4)=9 THEN CALL COMPLIMENT(2);
    IF SHR(R2(0),4)<>0 THEN OVERFLOW=TRUE;
END CHECK$RESULT;


CHECK$SIGN: RROCEDURE;
    IF SIGN0(0) AND SIGN0(1) THEN
    DO;
         SIGN0(2)=ROSITIVE;
         RETURN;
    END;
    SIGN0(2)=NEGITIVE;
    IF NOT SIGN0(0) AND NOT SIGN0(1) THEN RETURN;
    IF SIGN0(0) THEN CALL COMRLIMENT(1);
    ELSE CALL COMRLIMENT(0);
END CHECK$SIGN;


LEADING$ZEROES  RROCEDURE (ADDR) BYTE;
    DECLARE COUNT BYTE, ADDR ADDRESS;
    COUNT=0;
    BASE=ADDR;
    DO CTR=0 TO 9;
         IF (B$BYTE(CTR) AND 0F0H) <> 0 THEN RETURN COUNT;
         COUNT=COUNT + 1;
         IF (B$BYTE(CTR) AND 0FH) <> 0 THEN RETURN COUNT;
         COUNT=COUNT + 1;
    END;
    RETURN COUNT;
END LEADING$ZEROES;


CHECK$DECIMAL  RROCEDURE;
    IF DEC$RT2<>(CTR =C$BYTE(2)) THEN
    DO;
         BASE= R2;
         IF DEC$RT2 > CTR THEN CALL SHIFT$RIGHT(DEC$PT2-CTR);
         ELSE CALL SHIFT$LEFT(CTR-DEC$PT2);
    END;
    IF LEADING$ZEROES( R2) < 19 - C$BYTE(2) THEN OVERFLOW = TRUE;
END CHECK$DECIMAL;


ADD:  PROCEDURE;
    OVERFLOW=FALSE;
    CALL ALLIGN;
    CALL CHECK$SIGN;
    CALL ADDR0( R1, R2);
    CALL CHECK$RESULT;
END ADD;


ADD$SERIES  PROCEDURE(COUNT);
    DECLARE (I,COUNT) BYTE;
    DO I=1 TO COUNT;
         CALL ADD$R0( R2, R2);
```

```
            END,
        END ADD$SERIES,


        SET$MULT$DIV  PROCEDURE,
            OVERFLOW=FALSE,
            SIGN0(2) = (NOT (SIGN0(0) XOR SIGN0(1))) AND 01H;
            CALL FILL( R2, 10, 0),
        END SET$MULT$DIV;


        R1$GREATER. PROCEDURE BYTE,
            DECLARE I BYTE,
            DO CTR=0 TO 9;
                IF R1(CTR)><I =99H-R0(CTR)) THEN RETURN TRUE,
                IF R1(CTR)<I THEN RETURN FALSE,
            END,
            RETURN TRUE,
        END R1$GREATER;


        MULTIPLY  PROCEDURE(VALUE),
            DECLARE VALUE BYTE,
            IF VALUE<>0 THEN CALL ADD$SERIES(VALUE),
            BASE= R0;
            CALL ONE$LEFT,
        END MULTIPLY;


        DIVIDE. PROCEDURE,
            DECLARE (I, J, K, LZ0, LZ1, X) BYTE,
            CALL SET$MULT$DIV;
            IF    (LZ0 =LEADING$ZEROES(. R0))<>
                  (X   = (LZ1   = LEADING$ZERDES( R1))) THEN
             DO,
                IF LZ0>LZ1 THEN
                DO,
                    BASE =  R0;
                    CALL SHIFT$LEFT(I   = LZ0-LZ1),
                    DEC$PT0=DEC$PT0 + I,
                    X = LZ1,
                END,
                ELSE DO,
                    BASE = .R1;
                    CALL SHIFT$LEFT (I =LZ1-LZ0);
               •    DEC$PT1=DECPT1 + I,
                    X = LZ0;
                END,
            END,
            DECPT2= 16 - X + DECPT1 - DECPT0;
            CALL COMPLIMENT(0),
            DO I = X TO 19;
                J=0,
                DO WHILE R1$GREATER,
                    CALL ADD$R0( R1, R1),
                    IF R1(0) = 99H THEN
                            CALL COMPLIMENT (1),
                    J=J+1,
                END,
                K=SHR(I, 1),
                IF I THEN R2(K)=R2(K) OR J,
                ELSE R2(K)=R2(K) OR SHL(J, 4),
                BASE= R0;
                CALL ONE$RIGHT,
            END,
        END DIVIDE,




        LOAD$A$CHAR· PROCEDURE(CHAR),
            DECLARE CHAR BYTE,
            IF (SWITCH =NOT SWITCH) THEN
                B$BYTE(R$PTR)=B$BYTE(R$PTR) OR SHL(CHAR - 30H, 4),
            ELSE B$BYTE(R$PTR =R$PTR-1)=CHAR - 30H,
        END LOAD$A$CHAR,


        LOAD$NUMBERS  PROCEDURE(ADDR, CNT),
            DECLARE ADDR ADDRESS, (I, CNT)BYTE,
            HOLD=RES(ADDR),
            CTR=CNT,
            DO INDEX = 1 TO CNT;
                CTR=CTR-1,
                CALL LOAD$A$CHAR(H$BYTE(CTR)),
            END;
            CALL INC$PTR(5),
```

```
2        END LOADSNUMBERS,

1        SET$LOAD  PROCEDURE (SIGN$IN);
2          DECLARE SIGN$IN BYTE,'
2            DO CASE (CTR =C$BYTE(4)),
3                 BASE= P0;
3                 BASE= R1;
3                 BASE= R2,
3            END,
2            DEC$PTA(CTR)=C$BYTE(3),
2            SIGN0(CTR)=SIGN$IN;
2            CALL FILL (BASE, 10, 0),
2            R$PTR=9,
2            SWITCH=FALSE,
2        END SET$LOAD;

1        LOAD$NUMERIC  PROCEDURE,
2            CALL SET$LOAD(1),
2            CALL LOAD$NUMBERS(C$ADDR(0),C$BYTE(2));
2        END LOAD$NUMERIC,

1        LOAD$NUM$LIT  PROCEDURE,
2            DECLARE(LIT$SIZE,FLAG) BYTE,

2            CHAR$SIGN  PROCEDURE,
3                 LIT$SIZE=LIT$SIZE - 1,
3                 HOLD=HOLD + 1,
3            END CHAR$SIGN,

2            LIT$SIZE=C$BYTE(2),
2            HOLD=C$ADDR(0);
2            IF H$BYTE(0)='-' THEN
2            DO,
3                 CALL CHAR$SIGN,
3                 CALL SET$LOAD(NEGITIVE),
3            END,
2            ELSE DO;
3                 IF H$BYTE(0)='+' THEN CALL CHAR$SIGN,
3                 CALL SET$LOAD(POSITIVE);
3            END;
2            FLAG=0,
2            CTR=LIT$SIZE;
2            DO INDEX=1 TO LIT$SIZE,
3                 CTR=CTR-1,
3                 IF H$BYTE(CTR)='   THEN FLAG=LIT$SIZE - (CTR+1),
3                 ELSE CALL LOAD$A$CHAR(H$BYTE(CTR)),
3            END;
2            DEC$PTA(C$BYTE(4))= FLAG,
2            CALL INC$PTR(5);
2        END LOAD$NUM$LIT;

,1       STORE$ONE  PROCEDURE,
2            IF(SWITCH =NOT SWITCH) THEN
2                 B$BYTE(0)=SHR(H$BYTE(0), 4) OR '0',
2            ELSE DO,
3                 HOLD=HOLD-1,
3                 B$BYTE(0)=(H$BYTE(0) AND 0FH) OR '0',
3            END,
2            BASE=BASE-1,
2        END STORE$ONE,

1        STORE$AS$CHAR  PROCEDURE(COUNT),
2            DECLARE COUNT BYTE,
2            SWITCH=FALSE,
2            HOLD= R2 + 9;
2            DO CTR=1 TO COUNT,
3                 CALL STORE$ONE,
3            END,
2        END STORE$AS$CHAR,

1        SET$ZONE  PROCEDURE (ADDR),
2            DECLARE ADDR ADDRESS,
2            IF NOT SIGN0(2) THEN
2            DO,
3                 BASE=ADDR,
3                 B$BYTE(0)=B$BYTE(0) OR ZONE,
3            END,
2            CALL INC$PTR(4),
2        END SET$ZONE,
```

```
SET$SIGN$SEP   PROCEDURE (ADDR);
    DECLARE ADDR ADDRESS;
    BASE=ADDR;
    IF SIGNO(2) THEN B$BYTE(0)='+';
    ELSE B$BYTE(0)='-';
    CALL INC$PTR(4);
END SET$SIGN$SEP;


STORE$NUMERIC   PROCEDURE;
    CALL CHECK$DECIMAL;
    BASE=C$ADDR(0) + C$BYTE(2) -1;
    CALL STORE$AS$CHAR(C$BYTE(2));
END STORE$NUMERIC;



    /* * * * * * * * * * INPUT-OUTPUT ACTIONS * * * * * * * * * * * * */


DECLARE

FLAG$OFFSET         LIT        '33',
EXTENT$OFFSET       LIT        '12',
REC$NO              LIT        '32',
PTR$OFFSET          LIT        '17',
BUFF$LENGTH         LIT        '128',
VAR$END             LIT        'CR',
TERMINATOR          LIT              '1AH',
END$OF$RECORD       BYTE,
INVALID             BYTE,
RANDOM$FILE         BYTE,
CURRENT$FLAG        BYTE,
FCB$BYTE            BASED CURRENT$FCB          BYTE,
FCB$ADDR            BASED CURRENT$FCB          ADDRESS,
FCB$BYTE$A          BASED CURRENT$FCB (1) BYTE,
FCB$ADDR$A          BASED CURRENT$FCB (1) ADDRESS,
BUFF$PTR            ADDRESS,
BUFF$END            ADDRESS,
BUFFSTART           ADDRESS,
BUFF$BYTE           BASED BUFF$PTR       BYTE,
CON$BUFF            ADDRESS   INITIAL (80H),
CON$BYTE            BASED CON$BUFF       BYTE,
CON$INPUT           ADDRESS   INITIAL (82H);


ACCEPT   PROCEDURE;
    CALL CRLF;
    CALL PRINT$CHAR(3FH);
/*  CALL CRLF;   */
    CALL FILL(CON$INPUT,(CON$BYTE:=C$BYTE(2)),' ');
    CALL READ(CON$BUFF);
    CALL MOVE(CON$INPUT,RES(C$ADDR(0)),CON$BYTE);
    CALL INC$PTR(3);
END ACCEPT;


DISPLAY   PROCEDURE;
    DECLARE  B$CNT BYTE, BLANK LIT '20H';
    BASE=C$ADDR(0);
    CALL CRLF;
    B$CNT = C$BYTE(2);
    DO WHILE
        B$BYTE(B$CNT:=B$CNT - 1) = BLANK;
    END;
    DO CTR = 0 TO B$CNT;
        CALL PRINT$CHAR(B$BYTE(CTR));
    END;
    CALL INC$PTR(3);
END DISPLAY;


SET$FILE$TYPE   PROCEDURE(TYPE);
    DECLARE TYPE BYTE;
    BASE=C$ADDR(0);
    B$BYTE(FLAG$OFFSET)=TYPE;
END SET$FILE$TYPE;


GET$FILE$TYPE   PROCEDURE BYTE;
    BASE=C$ADDR(0);
    RETURN B$BYTE(FLAG$OFFSET);
END GET$FILE$TYPE;
```

```
SET$I$O: PROCEDURE;
     END$OF$RECORD, INVALID=FALSE;
     IF C$ADDR(0)=CURRENT$FCB THEN RETURN;
     /* STORE CURRENT POINTERS AND SET INTERNAL WRITE MARK */
     BASE=CURRENT$FCB;
     FCB$ADDR$A(PTR$OFFSET)=BUFF$PTR;
     FCB$BYTE$A(FLAG$OFFSET)=CURRENT$FLAG;
     /* LOAD NEW VALUES */
     BUFF$END=(BUFF$START =(CURRENT$FCB =C$ADDR(0))+START$OFFSET)
          + BUFF$LENGTH;
     CURRENT$FLAG=FCB$BYTE$A(FLAG$OFFSET);
     BUFF$PTR=FCB$ADDR$A(PTR$OFFSET);
END SET$I$O;


OPEN$FILE: PROCEDURE(TYPE);
     DECLARE TYPE BYTE;
     CALL SET$FILE$TYPE(TYPE);
     CTR=OPEN(CURRENT$FCB:=C$ADDR(0));
     DO CASE TYPE-1;
          /* INPUT */
          DO;
               IF CTR=255 THEN CALL PRINT$ERROR('NF');
               FCB$ADDR$A(PTR$OFFSET)=CURRENT$FCB+100H;
          END;
          /* OUTPUT */
          DO;
               CALL DELETE;
               CALL MAKE(C$ADDR(0));
               FCB$ADDR$A(PTR$OFFSET)=CURRENT$FCB+START$OFFSET-1;
          END;
          /* I-O */
          DO;
               IF CTR=255 THEN CALL FATAL$ERROR('NF');
               FCB$ADDR$A(PTR$OFFSET)=CURRENT$FCB + 100H;
          END;
     END;
     CURRENT$FCB=0;     /* FORCE A PARAMETER LOAD */
     CALL SET$I$O;
     CALL INC$PTR(2);
END OPEN$FILE;


WRITE$MARK: PROCEDURE BYTE;
     RETURN ROL(CURRENT$FLAG,1);
END WRITE$MARK;


SET$WRITE$MARK: PROCEDURE;
     CURRENT$FLAG=CURRENT$FLAG OR 80H;
END SET$WRITE$MARK;


WRITE$RECORD: PROCEDURE;
     IF NOT SHR(CURRENT$FLAG,1) THEN CALL FATAL$ERROR('WI');
     CALL SET$DMA;
     CURRENT$FLAG=CURRENT$FLAG AND 0FH;
     IF (CTR:=DISK$WRITE)=0 THEN RETURN;
     INVALID=TRUE;
END WRITE$RECORD;


READ$RECORD: PROCEDURE;
     CALL SET$DMA;
     IF WRITE$MARK THEN CALL WRITE$RECORD;
     IF (CTR:=DISK$READ)=0 THEN RETURN;
     IF CTR=1 THEN END$OF$RECORD=TRUE;
     ELSE INVALID=TRUE;
END READ$RECORD;


READ$BYTE: PROCEDURE BYTE;
     IF (BUFF$PTR:=BUFF$PTR + 1) >= BUFFEND THEN
     DO;
          CALL READ$RECORD;
          IF END$OF$RECORD THEN RETURN TERMINATOR;
          BUFF$PTR=BUFF$START;
     END;
     RETURN BUFF$BYTE;
END READ$BYTE;


WRITE$BYTE: PROCEDURE (CHAR);
     DECLARE CHAR BYTE;
     IF (BUFF$PTR:=BUFF$PTR+1) >= BUFF$END THEN
     DO;
          CALL WRITE$RECORD
```

```
3              BUFF$PTR=BUFF$START;
3          END;
2          CALL SET$WRITE$MARK;
2          BUFF$BYTE=CHAR;
2      END WRITE$BYTE;


1      WRITE$END$MARK:  PROCEDURE;
2          CALL WRITE$BYTE(CR);
2          CALL WRITE$BYTE(LF);
2      END WRITE$END$MARK;


1      READ$END$MARK:  PROCEDURE;
2          IF READ$BYTE<>CR THEN CALL PRINT$ERROR('EM');
2          IF READ$BYTE<>LF THEN CALL PRINT$ERROR('EM');
2      END READ$END$MARK;


1      READ$VARIABLE PROCEDURE;
2          CALL SET$I$O;
2          BASE=C$ADDR(1);
2          DO A$CTR=0 TO C$ADDR(2)-1;
3              IF (CTR :=(B$BYTE(A$CTR) :=READ$BYTE)) = VAR$END THEN
3              DO;
4                  CTR=READ$BYTE;
4                  RETURN;
4              END;
3              IF CTR=TERMINATOR THEN
3              DO;
4                  END$OF$RECORD=TRUE;
4                  RETURN;
4              END;
3          END;
2          CALL READ$END$MARK;
2      END READ$VARIABLE;


1      WRITE$VARIABLE:  PROCEDURE;
2          DECLARE COUNT ADDRESS;
2          CALL SET$I$O;
2          BASE=C$ADDR(1);
2          COUNT=C$ADDR(2);
2          DO WHILE(B$BYTE(COUNT :=COUNT-1)<>' ')AND (COUNT<>0);
3          END;
2          DO A$CTR=0 TO COUNT;
3              CALL WRITE$BYTE(B$BYTE(A$CTR));
3          END;
2          CALL WRITE$END$MARK;
2      END WRITE$VARIABLE;


1      READ$TO$MEMORY:  PROCEDURE;
2          CALL SET$I$O;
2          BASE=C$ADDR(1);
2          DO A$CTR=0 TO C$ADDR(2)-1;
3              IF (B$BYTE(A$CTR) :=READ$BYTE)=TERMINATOR THEN
3              DO;
4                  END$OF$RECORD=TRUE;
4                  RETURN;
4              END;
3          END;
2          CALL READ$END$MARK;
2      END READ$TO$MEMORY;


1      WRITE$FROM$MEMORY:  PROCEDURE;
2          CALL SET$I$O;
2          BASE=C$ADDR(1);
2          DO A$CTR=0 TO C$ADDR(2)-1;
3              CALL WRITE$BYTE(B$BYTE(A$CTR));
3          END;
2          CALL WRITE$END$MARK;
2      END WRITE$FROM$MEMORY;

           /* * * * * * * * * RANDOM I-O PROCEDURES * * * * * * * * */


1      SET$RANDOM$POINTER:  PROCEDURE;
           /*
           THIS PROCEDURE READS THE RANDOM KEY AND COMPUTES
           WHICH RECORD NEEDS TO BE AVAILABLE IN THE BUFFER
           THAT RECORD IS MADE AVAILABLE AND THE POINTERS
           SET FOR INPUT OR OUTPUT
           */
2          DECLARE (BYTE$COUNT,RECORD) ADDRESS;
```

```
                        EXTENT BYTE,
2               CALL SET$I$O,
2               BYTE$COUNT=(C$ADDR(2)+2)+CONVERT$TO$HEX(C$ADDR(3),C$BYTE(6)),
2               RECORD=SHR(BYTE$COUNT,7),
2               EXTENT=SHR(RECORD,7),
2               IF EXTENT<>FCB$BYTE$A(EXTENT$OFFSET) THEN
2               DO,
3               IF WRITE$MARK THEN CALL WRITE$RECORD,
3               CALL CLOSE(C$ADDR(0)),
3                       FCB$BYTE$A(EXTENT$OFFSET)=EXTENT,
3                       IF OPEN(C$ADDR(0))<>0 THEN
3                       DO,
4                               IF SHR(CURRENT$FLAG,1) THEN CALL MAKE(C$ADDR(0)),
4                               ELSE INVALID=TRUE,
4                       END,
3               END,
2               BUFF$PTR=(BYTE$COUNT AND 7FH) + BUFF$START -1,
2               IF FCB$BYTE$A(REC$NO)<>(CTR =LOW(RECORD)AND 7FH) THEN
2               DO,
3               FCB$BYTE$A(32)=CTR,
3                       CALL READ$RECORD,
3               END,
2       END SET$RANDOM$POINTER,


1       GET$REC$NUMBER  PROCEDURE,
2               DECLARE (RECNUM, K) ADDRESS,
                        (I,CNT) BYTE,
                        J(4) ADDRESS DATA (10000,1000,100,10),
                        BUFF(5) BYTE,

2               REC$NUM=SHL(FCB$BYTE$A(EXTENT$OFFSET),7)+FCB$BYTE$A(REC$NO),
2               DO I=0 TO 3,
3               CNT=0,
3                       DO WHILE REC$NUM>=(K =J(I)),
4                               REC$NUM=REC$NUM - K,
4                               CNT=CNT + 1,
4                       END,
3                       BUFF(I)=CNT + '0',
3               END,
2               BUFF(4)=REC$NUM+'0',
2               IF (I =C$BYTE(8))<=5 THEN
2               CALL MOVE( BUFF+4-I,C$ADDR(3),I),
2               ELSE DO,
3               CALL FILL(C$ADDR(0),I-5,'  '),
3                       CALL MOVE( BUFF,C$ADDR(3)+I-6, 5),
3               END,
2       END GET$REC$NUMBER,


1       WRITE$ZERO$RECORD  PROCEDURE,
2               DO A$CTR=1 TO C$ADDR(2),
3                       CALL WRITE$BYTE(0),
3               END,
2       END WRITE$ZERO$RECORD,


1       WRITE$RANDOM  PROCEDURE,
2               CALL SET$RANDOM$POINTER,
2               CALL WRITE$FROM$MEMORY,
2               CALL INC$PTR(9),
2       END WRITE$RANDOM,


1       BACK$ONE$RECORD  PROCEDURE,
2               CALL SET$I$O,
2       IF (BUFF$PTR =BUFF$PTR-(C$ADDR(2)+2))<=BUFF$START THEN RETURN,
2               BUFF$PTR=BUFF$END-(BUFF$START - BUFF$PTR),
2               IF (FCB$BYTE$A(REC$NO) =FCB$BYTE$A(REC$NO)-1)=255 THEN
2               DO,
3               FCB$BYTE$A(EXTENT$OFFSET)=FCB$BYTE$A(EXTENT$OFFSET)-1,
3                       IF OPEN(C$ADDR(0))<> 0 THEN
3                       DO,
4                       CALL PRINT$ERROR('CP'),
4                               INVALID=TRUE,
4                       END,
3                       FCB$BYTE$A(REC$NO)=127,
3               END,
2               CALL READ$RECORD,
2       END BACK$ONE$RECORD,


                /* * * * * * * * * * * * * * * MOVES * * * * * * * * * * * * * * */


1       INC$HOLD  PROCEDURE,
2               HOLD=HOLD + 1,
```

```
2            CTR=CTR + 1,
2        END INC$HOLD,


1        LOAD$INC  PROCEDURE,
2            H$BYTE(0)=B$BYTE(0),
2            BASE=BASE+1,
2            CALL INC$HOLD;
2        END LOAD$INC.


1        CHECK$EDIT  PROCEDURE(CHAR),
2            DECLARE CHAR BYTE,
2            IF (CHAR= 0') OR (CHAR='/') THEN CALL INC$HOLD,
2            ELSE IF CHAR= B' THEN
2            DO;
3                 H$BYTE(0)=' ',
3                 CALL INC$HOLD;
3            END.
2            ELSE IF CHAR='A' THEN
2            DO,
3                 IF NOT LETTER(B$BYTE(0)) THEN CALL PRINT$ERROR('IC');
3                 CALL LOAD$INC,
3            END,
2            ELSE IF CHAR='9' THEN
2            DO,
3                 IF NOT NUMERIC (B$BYTE(0)) THEN CALL PRINT$ERROR( IC').
3                 CALL LOAD$INC,
3            END,
2            ELSE CALL LOAD$INC,
2        END CHECK$EDIT,

             /* * * * * * * * * * * * MACHINE ACTIONS * *  * * * * * * * * * */


1        STOP  PROCEDURE,
2            CALL PRINT(.('END OF JOB $ )),
2            CALL BOOTER,
2        END STOP,


             /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

                         THE PROCEDURE BELOW CONTROLS THE EXECUTION OF THE CODE.
                         IT DECODES EACH OP-CODE AND PERFORMS THE ACTIONS

             ** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */


1        EXECUTE  PROCEDURE,
2            DO FOREVER,
3                 DO CASE GET$OP$CODE;

4                          ,       /* CASE ZERO NOT USED */

           /* 01  ADD */

4                          CALL ADD;

           /* 02  SUB */

4                          DO.
5                               CALL COMPLIMENT(0),
5                               IF SIGN0(0) THEN SIGN0(0)=NEGITIVE,
5                               ELSE SIGN0(0)=POSITIVE.
5                               CALL ADD,
5                          END,

           /* 03  MUL */

4                          DO,
5                               DECLARE I BYTE,
5                               CALL SET$MULT$DIV.
5                               DECPT1,DECPT2=DECPT1 + DECPT0,
5                               CALL ALLIGN,
5                               CALL MULTIPLY(SHR(R1(I =9)),4)),
5                               DO INDEX=1 TO 9;
6                                    CALL MULTIPLY(R1(I =I-1) AND 0FH),
6                                    CALL MULTIPLY(SHR(R1(I)),4)),
6                               END.
5                          END.

           /* 04  DIV */

4                          CALL DIVIDE,
           /* 05  NEG */
```

```
820   4                 BRANCH$FLAG=NOT BRANCH$FLAG;

            /* 06  STP */

821   4                 CALL STOP;

            /* 07  STI */

822   4                 CALL STORE$IMMEDIATE;


            /* 08  RND */

823   4                 DO;
824   5                      CALL STORE$IMMEDIATE;
825   5                      CALL FILL( R2,10,0);
826   5                      R2(9)=1;
827   5                      CALL ADD;
828   5                 END;

            /* 09  RET */

829   4                 DO;
830   5                      IF C$ADDR(0)<>0 THEN
831   5                      DO;
832   6                           A$CTR=C$ADDR(0);
833   6                           C$ADDR(0)=0;
834   6                           PROGRAM$COUNTER=A$CTR;
835   6                      END;
836   5                      ELSE CALL INC$PTR(2);
837   5                 END;

            /* 10  CLS */

838   4                 DO;
839   5                      CALL SET$I$O;
840   5                      IF WRITE$MARK THEN CALL WRITE$RECORD;
841   5                      CALL CLOSE(C$ADDR(0));
843   5                      CALL INC$PTR(2);
844   5                 END;

            /* 11  SER */

845   4                 DO;
846   5                      IF OVERFLOW  THEN PROGRAM$COUNTER = C$ADDR(0);
848   5                      ELSE CALL INC$PTR(2);
849   5                 END;
            /* 12  BRN */

850   4                 PROGRAM$COUNTER=C$ADDR(0);

            /* 13  OPN */

851   4                 CALL OPEN$FILE(1);

            /* 14  OP1 */

852   4                 CALL OPEN$FILE(2);

            /* 15  OP2 */

853   4                 CALL OPEN$FILE(3);

            /* 16  RGT */

854   4                 DO;
855   5                      IF NOT SIGNO(2) THEN
856   5                           BRANCH$FLAG=NOT BRANCH$FLAG;
857   5                      CALL COND$BRANCH(0);
858   5                 END;

            /* 17  RLT */

859   4                 DO;
860   5                      IF SIGNO(2) THEN
861   5                           BRANCH$FLAG=NOT BRANCH$FLAG;
862   5                      CALL COND$BRANCH(0);
863   5                 END;

            /* 18  REQ */

864   4                 DO;
865   5                      IF R2$ZERO THEN
866   5                           BRANCH$FLAG=NOT BRANCH$FLAG;
867   5                      CALL COND$BRANCH(0);
868   5                 END;
```

```
/* 19  INV */

                CALL INCR$OR$BRANCH(INVALID);

/* 20  EOR */

                CALL INCR$OR$BRANCH(END$OF$RECORD);

/* 21. ACC */

                CALL ACCEPT;

/* 22  DIS */

                CALL DISPLAY;

/* 23  STD */

                DO;
                     CALL DISPLAY;
                     CALL STOP;
                END;

/* 24  LDI */

                DO;
                     C$ADDR(2)=CONVERT$TO$HEX(C$ADDR(0),C$BYTE(2))+1;
                     CALL INC$PTR(3);
                END;

/* 25. DEC */

                DO;
                     IF C$ADDR(0)<>0 THEN C$ADDR(0)=C$ADDR(0)-1;
                     IF C$ADDR(0)=0 THEN PROGRAM$COUNTER=C$ADDR(1);
                     ELSE CALL INC$PTR(4);
                END;

/* 26  STO */

                DO;
                     CALL STORE$NUMERIC;
                     CALL INC$PTR(4);
                END;

/* 27. ST1 */

                DO;
                     CALL STORE$NUMERIC;
                     CALL SET$ZONE(C$ADDR(0)+C$BYTE(2)-1);
                END;

/* 28. ST2 */

                DO;
                     CALL STORE$NUMERIC;
                     CALL SET$ZONE(C$ADDR(0));
                END;

/* 29  ST3 */

                DO;
                     CALL CHECK$DECIMAL;
                     BASE=C$ADDR(0) + C$BYTE(2) - 1;
                     CALL STORE$AS$CHAR(C$BYTE(2) - 1);
                     CALL SET$SIGN$SEP(C$ADDR(0) + C$BYTE(2) -1);
                END;

/* 30  ST4 */

                DO;
                     CALL CHECK$DECIMAL;
                     BASE=C$ADDR(0) + C$BYTE(2);
                     CALL STORE$AS$CHAR(C$BYTE(2)-1);
                     CALL SET$SIGN$SEP(C$ADDR(0));
                END;

/* 31  ST5 */

                DO;
                     CALL CHECK$DECIMAL;
                     R0(3)=R2(3) OR SIGNO(2);
                     CALL MOVE( R2 + 3 - C$BYTE(2),C$ADDR(0),C$BYTE(2));
                     CALL INC$PTR(4);
                END;

/* 32  LOD */
```

```
4                       CALL LOAD$NUM$LIT,

            /* 33  LD1 */

4                       CALL LOAD$NUMERIC,

            /* 34  LD2 */

4                       DO,
5                           DECLARE I BYTE,
5                           HOLD=C$ADDR(0),
5                           IF CHECK$FOR$SIGN(CTR =H$BYTE(I =C$BYTE(2)-1)) THEN
5                           DO,
6                               CALL SET$LOAD(POSITIVE),
6                               I=I+1,
6                           END;
5                           ELSE DO,
6                               CALL SET$LOAD(NEGITIVE);
6                               CALL LOAD$A$CHAR(CTR-ZONE),
6                           END;
5                           CALL LOAD$NUMBERS(C$ADDR(0), I),
5                       END,

            /* 35  LD3 */

4                       DO,
5                           HOLD=C$ADDR(0),
5                           IF CHECK$FOR$SIGN(H$BYTE(0)) THEN
5                           DO;
6                               CALL SET$LOAD(POSITIVE),
6                               CALL LOAD$NUMBERS(C$ADDR(0), C$BYTE(2)),
6                           END,
5                           ELSE DO,
6                               CALL SET$LOAD(NEGITIVE),
6                               CALL LOAD$NUMBERS(C$ADDR(0)+1, C$BYTE(2)-1),
6                               CALL LOAD$A$CHAR(H$BYTE(0)-ZONE),
6                           END,
5                       END,

            /* 36  LD4 */

4                       DO,
5                           HOLD=C$ADDR(0);
5                           IF H$BYTE(C$BYTE(2) - 1) = '+' THEN
5                               CALL SET$LOAD(1),
5                           ELSE CALL SET$LOAD(0),
5                           CALL LOAD$NUMBERS(C$ADDR(0), C$BYTE(2) -1),
5                       END;

            /* 37  LD5 */

4                       DO;
5                           HOLD=C$ADDR(0),
5                           IF(H$BYTE(0)= '+') THEN CALL SET$LOAD(1),
5                           ELSE CALL SET$LOAD(0),
5                           CALL LOAD$NUMBERS(C$ADDR(0), C$BYTE(2)-1),
5                       END,

            /* 38  LD6 */

4                       DO,
5                           DECLARE I BYTE,
5                           HOLD=C$ADDR(0),
5                           CALL SET$LOAD(H$BYTE(I =C$BYTE(2)-1)),
5                           BASE=BASE + 9 - I;
5                           DO CTR = 0 TO I,
6                               B$BYTE(CTR)=H$BYTE(CTR),
6                           END,
5                           B$BYTE(CTR)=B$BYTE(CTR) AND 0F0H,
5                           CALL INC$PTR(5);
5                       END,

            /* 39  PER */

4                       DO,
5                           BASE=C$ADDR(1)+1,
5                           B$ADDR(0)=C$ADDR(2),
5                           PROGRAM$COUNTER=C$ADDR(0),
5                       END,

            /* 40  CNU */

4                       CALL COMP$NUM$UNSIGNED,

            /* 41  CNS *
```

```
                        CALL COMP$NUM$SIGN;

            /* 42  CAL */

                        CALL COMP$ALPHA;

            /* 43  RWS */

                        DO;
                        CALL BACK$ONE$RECORD;
                                CALL WRITE$FROM$MEMORY;
                                CALL INC$PTR(6);
                        END;

            /* 44  DLS */

                        DO;
                        CALL BACK$ONE$RECORD;
                                CALL WRITE$ZERO$RECORD;
                                CALL INC$PTR(6);
                        END;

            /* 45  RDF */

                        DO;
                                CALL READ$TO$MEMORY;
                                CALL INC$PTR(6);
                        END;

            /* 46  WTF */

                        DO;
                                CALL WRITE$FROM$MEMORY;
                                CALL INC$PTR(6);
                        END;

            /* 47  RVL */

                        CALL READ$VARIABLE;

            /* 48  WVL */

                        CALL WRITE$VARIABLE;

            /* 49  SCR */

                        DO;
                                SUBSCRIPT(C$BYTE(2))=
                                        CONVERT$TO$HEX(C$ADDR(0),C$BYTE(3));
                                CALL INC$PTR(4);
                        END;

            /* 50  SGT */

                        CALL STRING$COMPARE(1);

            /* 51  SLT */

                        CALL STRING$COMPARE(0);

            /* 52  SEQ */

                        CALL STRING$COMPARE(2);

            /* 53  MOV */

                        DO;
                                CALL MOVE(RES(C$ADDR(1)),RES(C$ADDR(0)),C$ADDR(2));
                                IF C$ADDR(3)<>0 THEN CALL
                                        FILL(RES(C$ADDR(1)) + C$ADDR(2),C$ADDR(3),' ');
                                CALL INC$PTR(8);
                        END;

            /* 54  RRS */

                        DO;
                                CALL READ$TO$MEMORY;
                                CALL GET$REC$NUMBER;
                                CALL INC$PTR(9);
                        END;

            /* 55  WRS */

                        DO;
                        CALL WRITE$FROM$MEMORY;
                                CALL GET$REC$NUMBER;
                                CALL INC$PTR(9);
```

```
                        END;

               /* 56  RRR */

                        DO;
                             CALL SET$RANDOM$POINTER;
                             CALL READ$TO$MEMORY;
                             CALL INC$PTR(3);
                        END;

               /* 57  WRR */

                        CALL WRITE$RANDOM;

               /* 58  RWR */

                        CALL WRITE$RANDOM;

               /* 59: DLR */

                        DO;
                             CALL SET$RANDOM$POINTER;
                             CALL WRITE$ZERO$RECORD;
                             CALL INC$PTR(9);
                        END;

               /* 60: MED */

                        DO;
                             CALL MOVE(C$ADDR(3),C$ADDR(0),C$ADDR(4));
                             BASE=C$ADDR(1);
                             HOLD=C$ADDR(D);
                             CTR=0;
                             DO WHILE (CTR<C$ADDR(1))AND(CTR<C$ADDR(4));
                                  CALL CHECK$EDIT(H$BYTE(0));
                             END;
                             IF CTR < C$ADDR(4) THEN
                                  CALL FILL(HOLD,C$ADDR(4)-CTR,' ');
                        END;

               /* 61: MNE */

                  ;       /*   NULL CASE   */


               /* 62. GDP */

                        DO;
                             DECLARE OFFSET BYTE;
                             OFFSET=CONVERT$TO$HEX(C$ADDR(1),C$BYTE(1)-1);
                             IF OFFSET > C$BYTE(D) + 1 THEN
                             DO;
                                  CALL PRINT$ERROR('GD');
                                  CALL INC$PTR(SHL(C$BYTE(0),1) + 5);
                             END;
                             ELSE PROGRAM$COUNTER=C$ADDR(OFFSET + 2);
                        END;

                   END; /* END OF CASE STATEMENT */
              END; /* END OF DO FOREVER */
         END EXECUTE;

              /* * * * * * * * * * * * PROGRAM EXECUTION STARTS HERE * * * * * * */

         BASE=CODE$START;
         PROGRAM$COUNTER=B$ADDR(0);
         CALL EXECUTE;
         END;


MODULE INFORMATION

     CODE AREA SIZE     = 1B8AH   7050D
     VARIABLE AREA SIZE = 00C1H    193D
     MAXIMUM STACK SIZE = 0016H     22D
     1542 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE PART2
OBJECT MODULE PLACED IN :F1:PART2.OBJ
COMPILER INVOKED BY:  PLM80 :F1:PART2.PLM


              $  PAGELENGTH(90)
  1           PART2        /* MODULE NAME */
                  DO;

                /*        COBOL COMPILER - PART 2               */


              /*  100H  = MODULE LOAD POINT  */

                   /*    GLOBAL DECLARATIONS AND LITERALS    */

  2    1      DECLARE LIT LITERALLY 'LITERALLY';
  3    1      DECLARE
                  HASH$TAB$ADDR  LIT        '1500H', /* ADDRESS OF THE BOTTOM.OF
                                                      THE TABLES  FROM PART1 */
                  PASS1$LEN      LIT        '46',
                  MAX$MEMORY     LIT        '3200H',
                  PASS1$TOP      LIT        '3100H',
                  CR             LIT        '13',
                  LF             LIT        '10',
                  QUOTE          LIT        '22H',
                  POUND          LIT        '23H',
                  TRUE           LIT        '1',
                  FALSE          LIT        '0',
                  FOREVER        LIT        'WHILE TRUE',
                  IF$FLAG        BYTE       INITIAL(FALSE);
  4    1      DECLARE MAXRNO LITERALLY '62',/* MAX READ COUNT */
                      MAXLNO LITERALLY '105',/* MAX LOOK COUNT */
                      MAXPNO LITERALLY '120',/* MAX PUSH COUNT */
                      MAXSNO LITERALLY '218',/* MAX STATE COUNT */
                      STARTS LITERALLY '1';/* START STATE */
  5    1      DECLARE READ1(*) BYTE
                   DATA(0, 63, 5, 6, 9, 14, 16, 20, 22, 24, 26, 31, 32, 41, 42, 44, 45, 49, 53
                  , 54, 58, 60, 46, 28, 46, 23, 28, 29, 36, 37, 48, 59, 11, 35, 46, 34, 13, 28, 29, 36, 37
                  , 48, 3, 1, 40, 23, 48, 57, 1, 56, 2, 30, 43, 27, 19, 33, 50, 52, 64, 18, 4, 38, 26, 39
              , 48
                  , 61, 55, 1, 15, 7, 12, 10, 51, 5, 9, 14, 16, 20, 22, 24, 26, 31, 41, 42, 44, 45, 49, 53,
              54
                  , 53, 60, 51, 7, 17, 1, 1, 5, 9, 14, 16, 20, 21, 22, 24, 26, 31, 41, 42, 44, 45, 49, 53
              , 54
                  , 58, 60, 48, 62, 3, 46, 25, 0, 0);
  6    1      DECLARE LOOK1(*) BYTE
                   DATA(0, 48, 0, 40, 0, 2, 0, 40, 0, 1, 15, 0, 48, 0, 30, 43, 0, 2, 0, 27, 0, 7
              , 0
                  , 17, 0, 1, 15, 0, 55, 0, 53, 0, 55, 0, 55, 0, 1, 15, 0, 12, 0, 1, 0, 51, 0, 48, 0, 25, 0, 6,
              48
                  , 0);
  7    1      DECLARE APPLY1(*) BYTE
                   DATA(0, 0, 22, 0, 6, 0, 0, 77, 0, 0, 81, 0, 11, 66, 68, 74, 79, 0, 0, 3, 81
              , 0
                  , 3, 81, 0, 35, 0, 0, 0, 0, 57, 58, 59, 0, 0, 0, 0, 0, 0, 0, 69, 0, 0, 0, 0, 0, 0, 5, 7, 8, 13,
              14
                  , 44, 0, 0, 2, 5, 6, 7, 8, 12, 13, 14, 10, 21, 23, 24, 26, 27, 28, 29, 33, 34, 40, 44, 75,
              76
                  , 77, 80, 0, 9, 30, 37, 38, 49, 51, 54, 0, 5, 7, 8, 13, 14, 28, 44, 0, 52, 0, 20, 0, 0, 15,
              32
                  , 61, 65, 0, 0, 0, 3, 81, 0, 0);
  8    1      DECLARE READ2(*) BYTE
                   DATA(0, 41, 6, 210, 9, 10, 63, 15, 17, 16, 20, 23, 24, 27, 28, 29, 30, 32
                  , 33, 34, 37, 38, 31, 201, 85, 84, 201, 205, 207, 206, 85, 178, 194, 192, 193, 185
              , 172
                  , 210, 205, 207, 206, 209, 202, 129, 26, 191, 197, 86, 3, 35, 4, 189, 188, 21, 167
              , 168
                  , 166, 161, 162, 14, 5, 181, 201, 25, 85, 39, 169, 2, 11, 7, 164, 174, 184, 6, 9, 10
              , 82
                  , 15, 17, 18, 20, 23, 27, 28, 29, 30, 32, 33, 34, 37, 38, 184, 5, 13, 130, 131, 6, 9, 10
                  , 83, 15, 16, 17, 18, 20, 23, 27, 28, 29, 30, 32, 33, 34, 37, 38, 198, 40, 121, 198, 19
              , 0
                  , 0);
  9    1      DECLARE LOOK2(*) BYTE
                   DATA(0, 12, 106, 22, 107, 198, 199, 36, 108, 142, 142, 124, 44, 109
              , 45
                  , 45, 110, 46, 196, 47, 111, 112, 49, 113, 52, 114, 114, 54, 56, 115, 57, 116, 58
              , 117
                  , 59, 118, 119, 119, 63, 64, 120, 147, 67, 69, 139, 75, 122, 78, 136, 128, 128, 81);
 10    1      DECLARE APPLY2(*) BYTE
                   DATA(0, 0, 137, 60, 76, 103, 77, 127, 126, 105, 73, 72, 151, 150, 152
                  , 177, 149, 132, 133, 104, 104, 136, 102, 101, 139, 132, 74, 160, 46, 65, 155, 153
                  , 156, 154, 143, 68, 134, 61, 94, 146, 66, 173, 79, 159, 55, 188, 60, 96, 144, 97, 98
                  , 95, 175, 135, 130, 42, 90, 67, 90, 90, 215, 90, 90, 117, 179, 138, 88, 124, 89, 90
                  , 157, 91, 158, 143, 90, 125, 125, 42, 145, 43, 91, 50, 51, 93, 101, 203, 53, 211
```

```
,195
    ,195, 195, 195, 195, 195, 195, 100, 71, 70, 208, 212, 171, 62, 99, 213, 163, 100
,140
    ,141, 101, 101, 147, 82),
DECLARE INDEX1(*) BYTE
     DATA(0, 1, 115, 2, 22, 115, 115, 115, 115, 33, 25, 73, 115, 115, 115,
26
    ,31, 32, 115, 35, 36, 115, 44, 115, 115, 26, 115, 115, 115, 115, 23, 42, 26, 115
,115
    ,43, 44, 23, 23, 45, 115, 47, 48, 50, 115, 51, 50, 53, 54, 23, 59, 60, 23, 61, 62, 65,
66
    ,66, 66, 66, 67, 68, 69, 26, 70, 26, 73, 71, 73, 91, 92, 93, 94, 95, 96, 115, 115, 117
    ,119, 73, 115, 2, 26, 1, 3, 5, 7, 9, 12, 14, 17, 19, 21, 23, 25, 28, 30, 32, 34, 36, 39,
41
    ,43, 45, 47, 49, 216, 123, 123, 176, 167, 180, 204, 204, 183, 170, 170, 170, 170
,214
    ,165, 1, 2, 2, 4, 4, 6, 6, 7, 7, 9, 9, 10, 10, 10, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
12
    ,12, 12, 12, 18, 18, 18, 18, 19, 19, 19, 19, 22, 22, 22, 25, 27, 27, 27, 28, 28, 29, 29
    ,29, 30, 30, 34, 34, 35, 35, 36, 36, 37, 37, 38, 38, 39, 39, 39, 40, 42, 43, 43, 44, 44
    ,45, 45, 46, 46, 46, 47, 47, 54, 55, 60, 60, 60, 68, 96, 96, 98, 98, 98, 100, 100, 100
    ,101, 101, 106, 106, 107, 107, 108, 111),
DECLARE INDEX2(*) BYTE
     DATA(0, 1, 1, 20, 1, 1, 1, 1, 1, 2, 1, 18, 1, 1, 1, 5, 1, 3, 1, 1, 6, 1, 1, 1,
1
    ,5, 1, 1, 1, 1, 2, 1, 5, 1, 1, 1, 1, 2, 2, 2, 1, 1, 2, 1, 1, 2, 1, 1, 5, 2, 1, 1, 2, 1, 3, 1, 1, 1
,1
    ,1, 1, 1, 1, 5, 1, 5, 18, 2, 16, 1, 1, 1, 1, 1, 19, 1, 2, 2, 1, 18, 1, 20, 5, 2, 2, 2, 1, 3, 2,
3
    ,2, 2, 2, 2, 3, 2, 2, 2, 2, 3, 2, 2, 2, 2, 2, 3, 12, 22, 36, 44, 45, 47, 49, 52, 54, 56, 57,
58
    ,59, 63, 64, 5, 1, 0, 0, 1, 0, 1, 2, 2, 1, 2, 0, 0, 2, 1, 0, 2, 1, 0, 2, 1, 1, 3, 3, 2, 3, 0, 1,
2
    ,2, 4, 2, 5, 4, 4, 5, 1, 1, 2, 2, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1
,0
    ,0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
,1
    ,0),

    /* END OF TABLES */
DECLARE
    /* JOINT DECLARATIONS */
    /* THE FOLLOWING ITEMS ARE DECLARED TOGETHER IN THIS
    GROUP IN ORDER TO FACILITATE THEIR BEING PASSED FROM
    THE FIRST PART OF THE COMPILER.
    */
    OUTPUT$FCB      (33) BYTE,
    DEBUGGING       BYTE,
    PRINT$PROD      BYTE,
    PRINT$TOKEN     BYTE,
    LIST$INPUT      BYTE,
    SEQ$NUM         BYTE,
    NEXT$SYM        ADDRESS,
    POINTER         ADDRESS,   /* POINTS TO THE NEXT BYTE TO BE READ */
    NEXT$AVAILABLE ADDRESS,
    MAX$INT$MEM    ADDRESS,

    /*  I O BUFFERS AND GLOBALS */
    IN$ADDR ADDRESS INITIAL (5CH),
    INPUTFCB BASED INADDR (33) BYTE,
    OUTPUT$BUFF     (128)          BYTE,
    OUTPUT$PTR              ADDRESS,
    OUTPUT$END              ADDRESS,
    OUTPUT$CHAR     BASED OUTPUT$PTR BYTE,

    /* MESSAGES FOR OUTPUT */

DECLARE
    ERROR$NEAR$$ (*) BYTE DATA (' ERROR NEAR $'),
    END$OF$PART$2(*) BYTE DATA (' END OF COMPILATION  $'),

    /* GLOBAL COUNTERS */
DECLARE
    CTR BYTE,
    A$CTR ADDRESS,
    BASE ADDRESS,
    B$BYTE BASED BASE BYTE,
    B$ADDR BASED BASE ADDRESS;

MON1  PROCEDURE (F, A) EXTERNAL;
    DECLARE F BYTE, A ADDRESS;
END MON1;

MON2  PROCEDURE (F, A) BYTE EXTERNAL;
    DECLARE F BYTE, A ADDRESS;
END MON2;
```

```
BOOT  PROCEDURE EXTERNAL,
          END BOOT,

PRINTCHAR  PROCEDURE (CHAR),
      DECLARE CHAR BYTE,
      CALL MON1 (2,CHAR),
END PRINTCHAR,

CRLF  PROCEDURE,
      CALL PRINTCHAR(CR),
      CALL PRINTCHAR(LF),
END CRLF,

PRINT  PROCEDURE (A),
      DECLARE A ADDRESS,
      CALL MON1 (9,A);
END PRINT;

PRINT$ERROR: PROCEDURE (CODE);
      DECLARE CODE ADDRESS,
      CALL CRLF,
      CALL PRINTCHAR(HIGH(CODE)),
      CALL PRINTCHAR(LOW(CODE)),
END PRINT$ERROR,

FATAL$ERROR  PROCEDURE(REASON),
      DECLARE REASON ADDRESS,
      CALL PRINT$ERROR(REASON),
      CALL TIME(10),
      CALL BOOT,
END FATAL$ERROR;

CLOSE  PROCEDURE,
      IF MON2(16, OUTPUT$FCB)=255 THEN CALL FATAL$ERROR('CL'),
END CLOSE,

MORE$INPUT  PROCEDURE BYTE,
      /* READS THE INPUT FILE AND RETURNS TRUE IF A RECORD
              WAS READ.  FALSE IMPLIES END OF FILE */
      DECLARE DCNT BYTE,
      IF (DCNT =MON2(20, INPUT$FCB))>1 THEN CALL FATAL$ERROR('BR'),
      RETURN NOT(DCNT),
END MORE$INPUT,

WRITE$OUTPUT  PROCEDURE (LOCATION),
      /* WRITES OUT A 128 BYTE BUFFER FROM LOCATION*/
      DECLARE LOCATION ADDRESS,
      CALL MON1(26,LOCATION); /* SET DMA */
      IF MON2(21, OUTPUT$FCB)<>0 THEN CALL FATAL$ERROR('WR'),
      CALL MON1(26,80H),  /*RESET DMA */
END WRITE$OUTPUT,

MOVE  PROCEDURE(SOURCE, DESTINATION, COUNT),
      /* MOVES FOR THE NUMBER OF BYTES SPECIFIED BY COUNT */
      DECLARE (SOURCE,DESTINATION) ADDRESS,
      (S$BYTE BASED SOURCE, D$BYTE BASED DESTINATION, COUNT) BYTE,
      DO WHILE (COUNT =COUNT - 1) <> 255,
              D$BYTE=S$BYTE,
              SOURCE=SOURCE +1,
              DESTINATION = DESTINATION + 1,
      END,
END MOVE,

FILL  PROCEDURE(ADDR,CHAR,COUNT),
      /* MOVES CHAR INTO ADDR FOR COUNT BYTES */
      DECLARE ADDR ADDRESS,
      (CHAR,COUNT,DEST BASED ADDR) BYTE,
      DO WHILE (COUNT =COUNT -1)<>255,
              DEST=CHAR;
              ADDR=ADDR + 1,
      END,
END FILL,

      /*    *   *   *   *   *  SCANNER LITS *   *   *   *   */
DECLARE
      LITERAL          LIT          '29',
      INPUT$STP        LIT          '48',
      PERIOD           LIT          '1',
      RPARIN           LIT          '3',
      LPARIN           LIT          '2',
      INVALID          LIT          '0',


      /*   * * * * SCANNER TABLES * * * *    */
DECLARE TOKEN$TABLE (*) BYTE DATA
      /* CONTAINS THE TOKEN NUMBER ONE LESS THAN THE FIRST RESERVED WORD
      FOR EACH LENGTH OF WORD  */
```

```
    (0, 0, 3, 7, 13, 29, 41, 48, 56, 60, 61),

TABLE (*) BYTE DATA('BY', 'GO', 'IF', 'TO', 'EOF', 'ADD', 'END', 'I-O'
    , 'NOT', 'RUN', 'CALL', 'ELSE', 'EXIT', 'FROM', 'INTO', 'LESS', 'MOVE'
    , 'NEXT', 'OPEN', 'PAGE', 'READ', 'SIZE', 'STOP', 'THRU', 'ZERO'
    , 'AFTER', 'CLOSE', 'ENTER', 'EQUAL', 'ERROR', 'INPUT', 'QUOTE', 'SPACE'
    , 'TIMES', 'UNTIL', 'USING', 'WRITE', 'ACCEPT', 'BEFORE', 'DELETE'
    , 'DIVIDE', 'OUTPUT', 'DISPLAY', 'GREATER'
    , 'INVALID', 'NUMERIC', 'PERFORM', 'REWRITE', 'ROUNDED', 'SECTION'
    , 'DIVISION', 'MULTIPLY', 'SENTENCE', 'SUBTRACT', 'ADVANCING',
    'DEPENDING', 'PROCEDURE', 'ALPHABETIC'),
OFFSET (11) ADDRESS INITIAL
    /*  NUMBER OF BYTES TO INDEX INTO THE TABLE FOR EACH LENGTH */
    (0, 0, 0, 8, 26, 86, 146, 176, 232, 264, 291),

WORD$COUNT (*) BYTE DATA
    /* NUMBER OF WORDS OF EACH SIZE */
    (0, 0, 4, 6, 15, 12, 5, 8, 4, 3, 1),


    MAX$ID$LEN      LIT         '12',
    MAX$LEN  .      LIT         '10',
    ADD$END        (*) BYTE DATA ('EOF   '),
    LOOKED          BYTE INITIAL (0),
    HOLD            BYTE,
    BUFFER$END      ADDRESS     INITIAL    (100H),
    NEXT          ' BASED POINTER          BYTE,
    INBUFF          LIT         '80H',
    CHAR            BYTE      INITIAL(' '),
    ACCUM (31)      BYTE,
    DISPLAY (74)    BYTE     INITIAL (0),
    TOKEN           BYTE,              /*RETURNED FROM SCANNER */


    /*  PROCEDURES USED BY THE SCANNER */

NEXT$CHAR: PROCEDURE BYTE;
    IF LOOKED THEN
    DO;
          LOOKED=FALSE;
          RETURN (CHAR =HOLD);
    END;
    IF (POINTER =POINTER + 1) >= BUFFER$END THEN
    DO;
         IF NOT MORE$INPUT THEN
          DO;
               BUFFER$END= MEMORY;
               POINTER= ADD$END;
          END;
          ELSE POINTER=INBUFF;
    END;
    RETURN (CHAR =NEXT);
END NEXT$CHAR;

GET$CHAR  PROCEDURE;
    /* THIS PROCEDURE IS CALLED WHEN A NEW CHAR IS NEEDED WITHOUT
    THE DIRECT RETURN OF THE CHARACTER*/
    CHAR=NEXT$CHAR;
END GET$CHAR;

DISPLAY$LINE: PROCEDURE;
    IF NOT LIST$INPUT THEN RETURN;
    DISPLAY(DISPLAY(0) + 1) = '$';
    CALL PRINT( DISPLAY(1));
    DISPLAY(0)=0;
END DISPLAY$LINE;

LOAD$DISPLAY  PROCEDURE;
    IF DISPLAY(0)<72 THEN
         DISPLAY(DISPLAY(0) =DISPLAY(0)+1)=CHAR;
    CALL GET$CHAR;
END LOAD$DISPLAY;

PUT  PROCEDURE;
    IF ACCUM(0) < 30 THEN
    ACCUM(ACCUM(0) =ACCUM(0)+1)=CHAR;
    CALL LOAD$DISPLAY;
END PUT;

EAT$LINE: PROCEDURE;
    DO WHILE CHAR<>CR;
         CALL LOAD$DISPLAY;
    END;
END EAT$LINE;

GET$NO$BLANK  PROCEDURE.
```

```
2          DECLARE (N, I) BYTE;
2          DO FOREVER;
3               IF CHAR =    ' THEN CALL LOAD$DISPLAY;
                ELSE
3               IF CHAR=CR THEN
3               DO;
4                    CALL DISPLAY$LINE;
4                    IF SEQ$NUM THEN N=8; ELSE N=2;
4                    DO I = 1 TO N;
5                         CALL LOAD$DISPLAY;
5                    END;
4                    IF CHAR = '*' THEN CALL EAT$LINE;
4               END;
                     ELSE
3                    IF CHAR = '.' THEN
3                    DO;
4                         IF NOT DEBUGGING THEN CALL EAT$LINE;
                          ELSE
4                         CALL LOAD$DISPLAY;
4                    END;
                ELSE
3               RETURN;
3          END;    /* END OF DO FOREVER */
2      END GET$NO$BLANK;

1      SPACE: PROCEDURE BYTE;
2          RETURN (CHAR=' ') OR (CHAR=CR);
2      END SPACE;

1      LEFT$PARIN: PROCEDURE BYTE;
2          RETURN CHAR = '(';
2      END LEFT$PARIN;

1      RIGHT$PARIN: PROCEDURE BYTE;
2          RETURN CHAR = ')';
2      END RIGHT$PARIN;

1      DELIMITER: PROCEDURE BYTE;
           /* CHECKS FOR A PERIOD FOLLOWED BY A SPACE OR CR*/
2          IF CHAR <> '.' THEN RETURN FALSE;
2          HOLD=NEXT$CHAR;
2          LOOKED=TRUE;
2          IF SPACE THEN
2          DO;
3               CHAR = ' ';
3               RETURN TRUE;
3          END;
2          CHAR='.';
2          RETURN FALSE;
2      END DELIMITER;

1      END$OF$TOKEN: PROCEDURE BYTE;
2          RETURN SPACE OR  DELIMITER OR LEFT$PARIN OR RIGHT$PARIN;
2      END END$OF$TOKEN;

1      GET$LITERAL: PROCEDURE BYTE;
2          CALL LOAD$DISPLAY;
2          DO FOREVER;
3              IF CHAR = QUOTE THEN
3                DO;
4                  CALL LOAD$DISPLAY;
4                  RETURN LITERAL;
4                END;
3              CALL PUT;
3          END;
2      END GET$LITERAL;


1      LOOK$UP: PROCEDURE BYTE;
2          DECLARE POINT ADDRESS,
           HERE BASED POINT (1) BYTE, I BYTE;

2          MATCH: PROCEDURE BYTE;
3               DECLARE J BYTE;
3               DO J=1 TO ACCUM(0);
4                    IF HERE(J - 1) <> ACCUM(J) THEN RETURN FALSE;
4               END;
3               RETURN TRUE;
3          END MATCH;

2          POINT=OFFSET(ACCUM(0))+ TABLE;
2          DO I=1 TO WORD$COUNT(ACCUM(0));
3               IF MATCH THEN RETURN I;
3               POINT = POINT + ACCUM(0);
3          END;
2          RETURN FALSE;
2      END LOOK$UP;
```

```
RESERVED$WORD:  PROCEDURE BYTE;
    /* RETURNS THE TOKEN NUMBER OF A RESERVED WORD IF THE CONTENTS OF
    THE ACCUMULATOR IS A RESERVED WORD, OTHERWISE RETURNS ZERO */
    DECLARE VALUE BYTE;
    DECLARE NUMB BYTE;
    IF ACCUM(0) <= MAX$LEN THEN
    DO;
         IF (NUMB :=TOKEN$TABLE(ACCUM(0)))<>0 THEN
         DO;
              IF (VALUE :=LOOK$UP) <> 0 THEN
                   NUMB=NUMB + VALUE;
              ELSE NUMB=0;
         END;
    END;
    RETURN NUMB;
END RESERVED$WORD;

GET$TOKEN:  PROCEDURE BYTE;
    ACCUM(0)=0;
    CALL GET$NO$BLANK;
    IF CHAR=QUOTE THEN RETURN GET$LITERAL;
    IF DELIMITER THEN
    DO;
         CALL PUT;
         RETURN PERIOD;
    END;
    IF LEFT$PARIN THEN
    DO;
         CALL PUT;
         RETURN LPARIN;
    END;
    IF RIGHT$PARIN THEN
    DO;
         CALL PUT;
         RETURN RPARIN;
    END;
    DO FOREVER;
         CALL PUT;
         IF END$OF$TOKEN THEN RETURN INPUT$STR;
    END; /* OF DO FOREVER */
END GET$TOKEN;

    /*    END OF SCANNER ROUTINES  */

    /*    SCANNER EXEC  */

SCANNER:  PROCEDURE;
    IF(TOKEN :=GET$TOKEN) = INPUT$STR THEN
         IF (CTR :=RESERVED$WORD) <> 0 THEN TOKEN=CTR;
END SCANNER;


PRINT$ACCUM:  PROCEDURE;
    ACCUM(ACCUM(0)+1)='$';
    CALL PRINT( .ACCUM(1));
END PRINT$ACCUM;

PRINT$NUMBER:  PROCEDURE(NUMB);
    DECLARE(NUMB, I, CNT, K) BYTE,  J (*) BYTE DATA(100,10);
    DO I=0 TO 1;
         CNT=0;
         DO WHILE NUMB >= (K :=J(I));
              NUMB=NUMB - K;
              CNT=CNT + 1;
         END;
         CALL PRINTCHAR('0' + CNT);
    END;
    CALL PRINTCHAR('0' + NUMB);
END PRINT$NUMBER;


    /*  *  *  *  END OF SCANNER PROCEDURES  *  *  *  */


    /*  *  *  *  *  SYMBOL TABLE DECLARATIONS  *  *  *  */

DECLARE

CUR$SYM             ADDRESS,          /*SYMBOL BEING ACCESSED*/
SYMBOL              BASED CUR$SYM (1)  BYTE,
SYMBOL$ADDR         BASED CUR$SYM (1)  ADDRESS,
NEXT$SYM$ENTRY      BASED NEXT$SYM      ADDRESS,
HASH$MASK           LIT        '3FH',
S$TYPE              LIT             '2',
DISPLACEMENT        LIT         12',
```

```
            OCCURS              LIT         '11',
            F$LENGTH            LIT               '2',
            FLD$LENGTH          LIT               '3',
            LEVEL               LIT               10',
            REL$ID              LIT               '5',
            LOCATION            LIT               '2',
            START$NAME          LIT               '11',   /*1 LESS*/
            FCB$ADDR            LIT         '4',

                 /* * * * * * * SYMBOL TYPE LITERALS * * * * * * */

            UNRESOLVED          LIT         '255',
            LABEL$TYPE          LIT         '32',
            MULT$OCCURS         LIT         '128',
            GROUP               LIT         '6',
            NON$NUMERIC$LIT     LIT         '7',
            ALPHA               LIT         '8',
            ALPHA$NUM           LIT         '9',
            LIT$SPACE           LIT         '10',
            LIT$QUOTE           LIT         '11',
            LIT$ZERO            LIT         '12',
            NUMERIC$LITERAL     LIT         '15',
            NUMERIC             LIT         '16',
            COMP                LIT         '21',
            A$ED                LIT         '72',
            A$N$ED              LIT         '73',
            NUM$ED              LIT         '80',

                /*   *   *   *   SYMBOL TABLE ROUTINES   *   *   *   */

       1    SET$ADDRESS  PROCEDURE(ADDR);
       2    DECLARE ADDR ADDRESS;
       2        SYMBOL$ADDR(LOCATION)=ADDR;
       2    END SET$ADDRESS;

       1    GET$ADDRESS  PROCEDURE ADDRESS;
       2        RETURN SYMBOL$ADDR(LOCATION);
       2    END GET$ADDRESS;

       1    GET$FCB$ADDR  PROCEDURE ADDRESS;
       2        RETURN SYMBOL$ADDR(FCB$ADDR);
       2    END GET$FCB$ADDR;

       1    GET$TYPE  PROCEDURE BYTE;
       2        RETURN SYMBOL(S$TYPE);
       2    END GET$TYPE;

       1    SET$TYPE  PROCEDURE(TYPE);
       2        DECLARE TYPE BYTE;
       2        SYMBOL(S$TYPE)=TYPE;
       2    END SET$TYPE;

       1    GET$LENGTH  PROCEDURE ADDRESS;
       2        RETURN SYMBOL$ADDR(FLD$LENGTH);
       2    END GET$LENGTH;

       1    GET$LEVEL  PROCEDURE BYTE;
       2        RETURN SHR(SYMBOL(LEVEL),4);
       2    END GET$LEVEL;

       1    GET$DECIMAL  PROCEDURE BYTE;
       2        RETURN SYMBOL(LEVEL) AND 0FH;
       2    END GET$DECIMAL;

       1    GET$P$LENGTH  PROCEDURE BYTE;
       2    RETURN SYMBOL(P$LENGTH);
       2    END GET$P$LENGTH;

       1    BUILD$SYMBOL  PROCEDURE(LEN);
       2        DECLARE LEN BYTE, TEMP ADDRESS;
       2        TEMP=NEXT$SYM;
       2        IF (NEXT$SYM = SYMBOL(LEN =LEN + DISPLACEMENT))
                    > MAX$MEMORY THEN CALL FATAL$ERROR('ST');
       2        CALL FILL (TEMP,0,LEN);
       2    END BUILD$SYMBOL;

       1    AND$OUT$OCCURS  PROCEDURE (TYPE$IN) BYTE;
       2        DECLARE TYPE$IN BYTE;
       2        RETURN TYPE$IN  AND 127;
       2    END AND$OUT$OCCURS;

                /*   *   *   *   PARSER DECLARATIONS   *   *   *   */
       1    DECLARE
```

```
PSTACKSIZE     LIT        '30',      /* SIZE OF PARSE STACKS*/
VALUE          (PSTACKSIZE)   ADDRESS,   /* TEMP VALUES */
STATESTACK     (PSTACKSIZE)   BYTE,   /* SAVED STATES */
VALUE2         (PSTACKSIZE)   ADDRESS,      /* VALUE2  STACK*/
VARC                (100)          BYTE,      /*TEMP CHAR STORE*/
IO$STACK       (20)       ADDRESS,
IO$PTR         BYTE,
MAX$BYTE       BASED          MAX$INT$MEM          BYTE,
SUB$INO        BYTE       INITIAL   (0),
CONO$TYPE      BYTE,
HOLD$SECTION   ADDRESS,
HOLD$SEC$ADDR  ADDRESS,
SECTION$FLAG   BYTE       INITIAL (0),
L$ADDR         ADDRESS,
L$LENGTH       ADDRESS,
L$TYPE         BYTE,
L$DEC          BYTE,
CON$LENGTH     BYTE,
COMPILING      BYTE    INITIAL(TRUE),
SP             BYTE    INITIAL (255),
MP             BYTE,
MPP1           BYTE,
NOLOOK         BYTE    INITIAL(FALSE),
(I, J, K)      BYTE,        /*INDICIES FOR THE PARSER*/
STATE          BYTE    INITIAL(STARTS),

    /* * * * * * * * CODE LITERALS * * * * * * * * * * */


    /* THE CODE LITERALS ARE BROKEN INTO GROUPS DEPENDING
    ON THE TOTAL LENGTH OF CODE PRODUCED FOR THAT ACTION */

    /* LENGTH ONE */
ADD LIT  '1', /* REGISTER ADDITION */
SUB LIT  '2', /* REGISTER SUBTRACTION */
MUL LIT  '3', /* REGISTER MULTIPLICATION */
DIV LIT  '4', /* REGISTER DIVISION */
NEG LIT  '5', /* NOT OPERATOR */
STP LIT  '6', /* STOP PROGRAM */
STI LIT  '7', /* STORE REGISTER 1 INTO REGISTER 0 */

    /* LENGTH TWO */
RND LIT  '8', /* ROUND CONTENTS OF REGISTER 1 */

    /* LENGTH THREE */
RET LIT  '9', /* RETURN */
CLS LIT  '10',      /* CLOSE */
SER LIT  '11',      /* SIZE ERROR */
BRN LIT  '12',      /* BRANCH */
OPN LIT  '13',      /* OPEN FOR INPUT */
OP1 LIT  '14',      /* OPEN FOR OUTPUT */
OP2 LIT  '15',      /* OPEN FOR I-O */
RGT LIT  '16',      /* REGISTER GREATER THAN */
RLT LIT  '17',      /* REGISTER LESS THAN */
REQ LIT  '18',      /* REGISTER EQUAL */
INV LIT  '19',      /* INVALID FILE ACTION */
EOR LIT  '20',      /* END OF FILE REACHED */

    /* LENGTH FOUR */
ACC LIT  '21',      /* ACCEPT */
DIS LIT  '22',      /* DISPLAY */
STD LIT  '23',      /* STOP AND DISPLAY */
LDI LIT  '24',      /* LOAD COUNTER IMEDIATE */

    /* LENGTH FIVE */
DEC LIT  '25',      /* DECREMENT AND BRANCH IF ZERO */
STO LIT  '26',      /* STORE NUMERIC */
ST1 LIT  '27',      /* STORE SIGNED NUMERIC TRAILING */
ST2 LIT  '28',      /* STORE SIGNED NUMERIC LEADING */
ST3 LIT  '29',      /* STORE SEPARATE SIGN LEADING */
ST4 LIT  '30',      /* STORE SEPARATE SIGN TRAILING */
ST5 LIT  '31',      /* STORE COMPUTATIONAL */

    /* LENGTH SIX */
LOD LIT  '32',      /* LOAD NUMERIC LITERAL */
LD1 LIT  '33',      /* LOAD NUMERIC */
LD2 LIT  '34',      /* LOAD SIGNED NUMERIC TRAILING */
LD3 LIT  '35',      /* LOAD SIGNED NUMERIC LEADING */
LD4 LIT  '36',      /* LOAD SEPARATE SIGN TRAILING */
LD5 LIT  '37',      /* LOAD SEPARATE SIGN LEADING */
LD6 LIT  '38',      /* LOAD COMPUTATIONAL */

    /* LENGTH SEVEN */
PER LIT  '39',      /* PERFORM */
CNU LIT  '40',      /* COMPARE FOR UNSIGNED NUMERIC */
CNS LIT  '41',      /* COMPARE FOR SIGNED NUMERIC */
CAL LIT  '42',      /* COMPARE FOR ALPHABETIC */
```

```
          RWS LIT  '43',       /* REWRITE SEQUENTIAL */
          DLS LIT  '44',       /* DELETE SEQUENTIAL */
          RDF LIT  '45',       /* READ SEQUENTIAL */
          WTF LIT  '46',       /* WRITE SEQUENTIAL */
          RVL LIT  '47',       /* READ VARIABLE LENGTH */
          WVL LIT  '48',       /* WRITE VARIABLE LENGTH */

              /* LENGTH NINE */
          SCR LIT  '49',       /* SUBSCRIPT COMPUTATION */
          SGT LIT  '50',       /* STRING GREATER THAN */
          SLT LIT  '51',       /* STRING LESS THAN */
          SEQ LIT  '52',       /* STRING EQUAL */
          MOV LIT  '53',       /* MOVE */

              /* LENGTH 10 */
          RRS LIT  '54',       /* READ RELATIVE SEQUENTIAL */
          WRS LIT  '55',       /* WRITE RELATIVE SEQUENTIAL */
          RRR LIT  '56',       /* READ RELATIVE RANDOM */
          WRR LIT  '57',       /* WRITE RELATIVE RANDOM */
          RWR LIT  '58',       /* REWRITE RELATIVE */
          DLR LIT  '59',       /* DELETE RELATIVE */

              /* LENGTH ELEVEN */
          MED LIT  '60',       /* MOVE EDITED */

              /* LENGTH THIRTEEN */
          MNE LIT  '61',       /* MOVE NUMERIC EDITED */

              /* VARIABLE LENGTH */
          GDP LIT  '62',       /* GO DEPENDING ON */

              /* BUILD DIRECTING ONLY */
          INT LIT  '63',       /* INITIALIZE STORAGE */
          BST LIT  '64',       /* BACK STUFF ADDRESS */
          TER LIT  '65',       /* TERMINATE BUILD */
          SCD LIT  '66',       /* SET CODE START */

              /*   *   *   *   PARSER ROUTINES   *   *   *   *   */

  1       DIGIT  PROCEDURE (CHAR) BYTE,
  2           DECLARE CHAR BYTE,
  2           RETURN (CHAR<='9') AND (CHAR>='0');
  2       END DIGIT,

  1       LETTER  PROCEDURE BYTE;
  2           RETURN (CHAR>='A') AND (CHAR<='Z');
  2       END LETTER;


  1       INVALID$TYPE. PROCEDURE,
  2           CALL PRINT$ERROR('IT');
  2       END INVALID$TYPE;

  1       BYTE$OUT  PROCEDURE(ONE$BYTE),
  2           DECLARE ONE$BYTE BYTE,
  2           IF (OUTPUT$PTR =OUTPUT$PTR + 1) > OUTPUT$END THEN
  2           DO;
  3                CALL WRITE$OUTPUT(. OUTPUT$BUFF);
  3                OUTPUT$PTR= OUTPUT$BUFF,
  3           END;
  2           OUTPUT$CHAR=ONE$BYTE,
  2       END BYTE$OUT,

  1       ADDR$OUT: PROCEDURE (ADDR);
  2           DECLARE ADDR ADDRESS,
  2           CALL BYTE$OUT(LOW(ADDR)),
  2           CALL BYTE$OUT(HIGH (ADDR));
  2       END ADDR$OUT,

  1       INC$COUNT  PROCEDURE(CNT);
  2           DECLARE CNT BYTE,
  2           IF(NEXT$AVAILABLE =NEXT$AVAILABLE + CNT)
                   >MAX$INT$MEM THEN CALL FATAL$ERROR('MO'),
  2       END INC$COUNT,


  1       ONE$ADDR$OPP  PROCEDURE(CODE, ADDR),
  2           DECLARE CODE BYTE, ADDR ADDRESS,
  2           CALL BYTE$OUT(CODE),
  2           CALL ADDR$OUT(ADDR),
  2           CALL INC$COUNT(3);
  2       END ONE$ADDR$OPP,

  1       NOT$IMPLIMENTED  PROCEDURE,
  2           CALL PRINT$ERROR ('NI'),
  2       END NOT$IMPLIMENTED;
```

```
1        MATCH  PROCEDURE ADDRESS,
             /* CHECKS AN IDENTIFIER TO SEE IF IT IS IN THE SYMBOL
             TABLE.  IF IT IS PRESENT, CUR$SYM IS SET FOR ACCESS,
             OTHERWISE THE POINTERS ARE SET FOR ENTRY*/
2            DECLARE POINT ADDRESS, COLLISION BASED POINT ADDRESS, (HOLD, I) BYTE,
2            IF VARC(0)>MAX$ID$LEN THEN VARC(0)=MAX$ID$LEN;
2            HOLD=0;
2            DO I=1 TO VARC(0);
3                HOLD=HOLD+VARC(I);
3            END;
2            POINT=HASH$TAB$ADDR + SHL((HOLD AND HASH$MASK), 1);
2            DO FOREVER;
3                IF COLLISION=0 THEN
3                 DO,
4                      CUR$SYM, COLLISION=NEXT$SYM;
4                      CALL BUILD$SYMBOL(VARC(0)),
4                      SYMBOL(P$LENGTH)=VARC(0),
4                      DO I=1 TO VARC(0),
5                           SYMBOL(START$NAME+I)=VARC(I),
5                      END;
4                      CALL SET$TYPE(UNRESOLVED), /* UNRESOLVED LABEL */
4                      RETURN CUR$SYM;
4                 END,
                  ELSE
3                 DO,
4                      CUR$SYM=COLLISION,
4                      IF (HOLD =GET$P$LENGTH)=VARC(0) THEN
4                      DO;
5                           I=1,
5                           DO WHILE SYMBOL(START$NAME + I)= VARC(I),
6                            IF (I =I+1))HOLD THEN RETURN(CUR$SYM =COLLISION),
6                           END,
5                     END;
4                END;
3                POINT=COLLISION;
3            END,
2        END MATCH;

1        SET$VALUE. PROCEDURE(NUMB),
2            DECLARE NUMB ADDRESS,
2            VALUE(MP)=NUMB;
2        END SET$VALUE.

1        SET$VALUE2: PROCEDURE(ADDR),
2            DECLARE ADDR ADDRESS;
2            VALUE2(MP)=ADDR,
2        END SET$VALUE2;


1        SUB$CNT  PROCEDURE BYTE,
2            IF (SUB$IND =SUB$IND + 1)>6 THEN
2                 SUB$IND=1,
2            RETURN SUB$IND;
2        END SUB$CNT,


1        CODE$BYTE  PROCEDURE (CODE);
2            DECLARE CODE BYTE,
2            CALL BYTE$OUT(CODE),
2            CALL INC$COUNT(1),
2        END CODE$BYTE;


1        CODE$ADDRESS. PROCEDURE (CODE),
2            DECLARE CODE ADDRESS,
2            CALL ADDR$OUT(CODE),
2            CALL INC$COUNT(2);
2        END CODE$ADDRESS,


1        INPUT$NUMERIC  PROCEDURE BYTE,
2            DO CTR=1 TO VARC(0);
3                 IF NOT DIGIT(VARC(CTR)) THEN RETURN FALSE,
3            END;
2            RETURN TRUE;
2        END INPUT$NUMERIC,


1        CONVERT$INTEGER  PROCEDURE ADDRESS,
2            ACTR=0;
2            DO CTR=1 TO VARC(0);
3                 IF NOT DIGIT(VARC(CTR)) THEN CALL PRINT$ERROR('NN'),
3                 A$CTR=SHL(ACTR,3)+SHL(ACTR,1) + VARC(CTR) - '0',
3            END;
2            RETURN ACTR;
2        END CONVERT$INTEGER,
```

```
 1      BACK$STUFF: PROCEDURE (ADD1,ADD2);
 2          DECLARE (ADD1,ADD2) ADDRESS;
 2          CALL BYTE$OUT(BST);
 2          CALL ADDR$OUT(ADD1);
 2          CALL ADDR$OUT(ADD2);
 2      END BACK$STUFF;


 1      UNRESOLVED$BRANCH: PROCEDURE;
 2          CALL SET$VALUE(NEXT$AVAILABLE + 1);
 2          CALL ONE$ADDR$OPP(BRN,0);
 2          CALL SET$VALUE2(NEXT$AVAILABLE);
 2      END UNRESOLVED$BRANCH;


 1      BACK$COND: PROCEDURE;
 2          CALL BACK$STUFF(VALUE(SP-1),NEXT$AVAILABLE);
 2      END BACK$COND;


 1      SET$BRANCH: PROCEDURE;
 2          CALL SET$VALUE(NEXT$AVAILABLE);
 2          CALL CODE$ADDRESS(0);
 2      END SET$BRANCH;


 1      KEEP$VALUES: PROCEDURE;
 2          CALL SET$VALUE(VALUE(SP));
 2          CALL SET$VALUE2(VALUE2(SP));
 2      END KEEP$VALUES;


 1      STD$ATTRIBUTES: PROCEDURE(TYPE);
 2           DECLARE TYPE BYTE;
 2           CALL CODE$ADDRESS(GET$FCB$ADDR);
 2           CALL CODE$ADDRESS(GET$ADDRESS);
 2           CALL CODE$ADDRESS(GET$LENGTH);
 2           IF TYPE=0 THEN RETURN;
 2           CUR$SYM=SYMBOL$ADDR(REL$ID);
 2           CALL CODE$ADDRESS(GET$ADDRESS);
 2           CALL CODE$BYTE(GET$LENGTH);
 2      END STD$ATTRIBUTES;


 1      READ$WRITE: PROCEDURE(INDEX);
 2          DECLARE INDEX BYTE;

 2           IF (CTR:=GET$TYPE)=1 THEN
 2          DO;
 3                CALL CODE$BYTE(RDF+INDEX);
 3                CALL STD$ATTRIBUTES(0);
 3          END;
 2          ELSE IF CTR=2 THEN
 2          DO;
 3                CALL CODE$BYTE(RRS+INDEX);
 3                CALL STD$ATTRIBUTES(1);
 3          END;
 2          ELSE IF CTR=3 THEN
 2          DO;
 3                CALL CODE$BYTE(RRR+INDEX);
 3                CALL STD$ATTRIBUTES(1);
 3          END;
 2          ELSE IF CTR=4 THEN
 2          DO;
 3                CALL CODE$BYTE(RVL+INDEX);
 3                CALL STD$ATTRIBUTES(0);
 3          END;
 2          ELSE CALL PRINT$ERROR('FT');
 2      END READ$WRITE;


 1      ARITHMETIC$TYPE: PROCEDURE BYTE;
 2          IF ((L$TYPE:=AND$OUT$OCCURS(L$TYPE))>=NUMERIC$LITERAL)
                AND (L$TYPE<=COMP) THEN RETURN L$TYPE - NUMERIC$LITERAL;
 2          CALL INVALID$TYPE;
 2          RETURN 0;
 2      END ARITHMETIC$TYPE;


 1      DEL$RNT: PROCEDURE(FLAG);
 2          DECLARE FLAG BYTE;
 2          IF (CTR:=GET$TYPE)=3 THEN
 2          DO;
 3               IF FLAG THEN CALL CODE$BYTE(RWR);
 3               ELSE CALL CODE$BYTE(DLR);
```

```
                    CALL STI$ATTRIBUTES(1);
                    RETURN;
               END;
               IF (CTR=2) AND (NOT FLAG) THEN CALL CODE$BYTE(DLS);
               ELSE IF (CTR<4) AND FLAG THEN CALL CODE$BYTE(RWS);
               ELSE CALL INVALID$TYPE;
               CALL STI$ATTRIBUTES(0);
          END DEL$RWT;


          ATTRIBUTES: PROCEDURE;
               CALL CODE$ADDRESS(L$ADDR);
               CALL CODE$BYTE(L$LENGTH);
               CALL CODE$BYTE(L$DEC);
          END ATTRIBUTES;


          LOAD$L$ID: PROCEDURE(S$PTR);

               DECLARE S$PTR BYTE;
               IF ((A$CTR := VALUE(S$PTR)) = NON$NUMERIC$LIT) OR
                    (ACTR = NUMERIC$LITERAL) THEN
               DO;
                    L$ADDR=VALUE2(S$PTR);
                    L$LENGTH=CON$LENGTH;
                    L$TYPE=A$CTR;
                    RETURN;
               END;
               IF A$CTR<=LIT$ZERO THEN
               DO;
                    L$TYPE, L$ADDR=A$CTR;
                    L$LENGTH=1;
                    RETURN;
               END;
               CUR$SYM=VALUE(S$PTR);
               L$TYPE=GET$TYPE;
               L$LENGTH=GET$LENGTH;
               L$DEC=GET$DECIMAL;
               IF(L$ADDR:=VALUE2(S$PTR))=0 THEN L$ADDR=GET$ADDRESS;
          END LOAD$L$ID;


          LOAD$REG: PROCEDURE(REG$NO,PTR);
               DECLARE (REG$NO,PTR) BYTE;
               CALL LOAD$L$ID(PTR);
               CALL CODE$BYTE(LOD+ARITHMETIC$TYPE);
               CALL ATTRIBUTES;
               CALL CODE$BYTE(REG$NO);
          END LOAD$REG;


          STORE$REG: PROCEDURE(PTR);
               DECLARE PTR BYTE;
               CALL LOAD$L$ID(PTR);
               CALL CODE$BYTE(STO + ARITHMETIC$TYPE -1);
               CALL ATTRIBUTES;
          END STORE$REG;


          STORE$CONSTANT: PROCEDURE ADDRESS;
               IF(MAX$INT$MEM:=MAX$INT$MEM - VARC(0))<NEXT$AVAILABLE
                    THEN CALL FATAL$ERROR('MO');
               CALL BYTE$OUT(INT);
               CALL ADDR$OUT(MAX$INT$MEM);
               CALL ADDR$OUT(CON$LENGTH:=VARC(0));
               DO CTR = 1 TO CON$LENGTH;
                    CALL BYTE$OUT(VARC(CTR));
               END;
               RETURN MAX$INT$MEM;
          END STORE$CONSTANT;


          NUMERIC$LIT: PROCEDURE BYTE;
               DECLARE CHAR BYTE;
               DO CTR=1 TO VARC(0);
                    IF NOT( DIGIT(CHAR:=VARC(CTR))
                         OR (CHAR='-') OR (CHAR='+')
                         OR (CHAR='.')) THEN RETURN FALSE;
               END;
               RETURN TRUE;
          END NUMERIC$LIT;


          ROUND$STORE: PROCEDURE;
               IF VALUE(SP)<>0 THEN
               DO;
                    CALL CODE$BYTE(RND);
```

```
3                 CALL CODE$BYTE(L$DEC)
3             END,
2             CALL STORE$REG(SP-1),
2         END ROUND$STORE,

1         ADD$SUB  PROCEDURE (INDEX);
2              DECLARE INDEX BYTE,
2             CALL LOAD$REG(0,MPP1),
2             IF VALUE(SP-1)<>0 THEN
2             DO,
3                 CALL LOAD$REG(1,SP-3),
3                 CALL CODE$BYTE(ADD);
3                 CALL CODE$BYTE(STI),
3             END,
2             CALL LOAD$REG(1,SP-1);
2             CALL CODE$BYTE(ADD + INDEX),
2             CALL ROUND$STORE,
2         END ADD$SUB,

1         MULT$DIV  PROCEDURE(INDEX);
2             DECLARE INDEX BYTE,
2             CALL LOAD$REG(0,MPP1),
2             CALL LOAD$REG(1,SP-1),
2             CALL CODE$BYTE(MUL + INDEX),
2             CALL ROUND$STORE,
2         END MULT$DIV;

1         CHECK$SUBSCRIPT  PROCEDURE,
2             CUR$SYM=VALUE(MP),
2             IF GET$TYPE<MULT$OCCURS THEN
2             DO,
3                 CALL PRINT$ERROR('IS'),
3                 RETURN,
3           . END,
2             IF INPUT$NUMERIC THEN
2             DO;
3                 CALL SET$VALUE2(GET$ADDRESS + (GET$LENGTH * CONVERT$INTEGER)),
3                 RETURN;
3             END;
2             CUR$SYM=MATCH,
2             IF ((CTR:=GET$TYPE)<NUMERIC) OR (CTR>COMP) THEN    _
2                 CALL PRINT$ERROR('TE'),
2 .           CALL ONE$ADDR$OPP(SCR,GET$ADDRESS);
2             CALL CODE$BYTE(SUB$CNT),
2             CALL CODE$BYTE(GET$LENGTH),
2             CALL SET$VALUE2(SUB$IND);
2         END CHECK$SUBSCRIPT,

1         LOAD$LABEL. PROCEDURE,
2             CUR$SYM=VALUE(MP),
2             IF (A$CTR =GET$ADDRESS)<>0 THEN
2                 CALL BACK$STUFF(A$CTR,VALUE2(MP)),
2             CALL SET$ADDRESS(VALUE2(MP)),
2             CALL SET$TYPE(LABEL$TYPE),
2             IF (A$CTR =GET$FCB$ADDR)<>0 THEN
2                     CALL BACK$STUFF(A$CTR,NEXT$AVAILABLE),
2             SYMBOL$ADDR(FCB$ADDR)=NEXT$AVAILABLE,
2             CALL ONE$ADDR$OPP(RET,0);
2         END LOAD$LABEL,

1         LOAD$SEC$LABEL. PROCEDURE,
2             A$CTR=VALUE(MP),
2             CALL SET$VALUE(HOLD$SECTION);
2             HOLD$SECTION=A$CTR,
2             A$CTR=VALUE2(MP);
2             CALL SET$VALUE2(HOLD$SEC$ADDR);
2             HOLD$SEC$ADDR = A$CTR,
2             CALL LOAD$LABEL,
2         END LOAD$SEC$LABEL,

1         LABEL$ADDR$OFFSET  PROCEDURE (ADDR, HOLD, OFFSET) ADDRESS,
2             DECLARE ADDR ADDRESS,
2             DECLARE (HOLD, OFFSET, CTR) BYTE,
2             CUR$SYM=ADDR;
2             IF(CTR =GET$TYPE)=LABEL$TYPE THEN
2             DO,
3                 IF HOLD THEN RETURN GET$ADDRESS,
3                 RETURN GET$FCB$ADDR,
3             END;
2             IF CTR<>UNRESOLVED THEN CALL INVALID$TYPE,
2             IF HOLD THEN
```

```
2              DO;
3                   A$CTR=GET$ADDRESS,
3                   CALL SET$ADDRESS(NEXT$AVAILABLE + OFFSET),
3                   RETURN A$CTR;
3              END,
2              A$CTR=GET$FCB$ADDR;
2              SYMBOL$ADDR(FCB$ADDR)=NEXT$AVAILABLE + OFFSET,
2              RETURN A$CTR;
2         END LABEL$ADDR$OFFSET,


1         LABEL$ADDR: PROCEDURE (ADDR, HOLO) ADDRESS;
2              DECLARE ADDR ADDRESS,
                       HOLD BYTE,
2              RETURN LABEL$ADDR$OFFSET (ADDR, HOLD, 1),
2         ENO LABEL$ADDR;


1         CODE$FOR$OISPLAY  PROCEOURE (POINT);
2              DECLARE POINT BYTE,
2              CALL LOAD$L$IO(POINT);
2              CALL ONE$ADDR$OPP(OIS, L$ADDR);
2              CALL CODE$BYTE(L$LENGTH),
2         ENO CODE$FOR$DISPLAY,


1         A$AN$TYPE  PROCEDURE BYTE,
2          RETURN (L$TYPE=ALPHA) OR (L$TYPE=ALPHA$NUM),
2         END A$AN$TYPE,


1         NOT$INTEGER  PROCEDURE BYTE,
2              RETURN L$DEC<>0;
2         ENO NOT$INTEGER;


1         NUMERIC$TYPE  PROCEDURE BYTE,
2              RETURN (L$TYPE>=NUMERIC$LITERAL) ANO (L$TYPE<=COMP),
2         END NUMERIC$TYPE,


1         GEN$COMPARE  PROCEDURE,
2              OECLARE (H$TYPE, H$OEC) BYTE,
                    (H$ADDR, H$LENGTH) AODRESS,

2              CALL LOAO$L$ID(MP);
2              L$TYPE=AND$OUT$OCCURS(L$TYPE),
2              IF COND$TYPE=3 THEN  /* COMPARE FOR NUMERIC */
2              DO;
3                   IF A$AN$TYPE OR (L$TYPE>COMP) THEN CALL INVALID$TYPE,
3                   CALL SET$VALUE2(NEXT$AVAILABLE),
3                   IF L$TYPE=NUMERIC THEN CALL CODE$BYTE(CNU);
3                   ELSE CALL CODE$BYTE(CNS),
3                   CALL CODE$AODRESS(L$ADOR),
3                   CALL CODE$ADDRESS(L$LENGTH),
3                   CALL SET$BRANCH,
3               END,
2              ELSE IF COND$TYPE=4 THEN
2              DO;
3                   IF NUMERIC$TYPE THEN CALL INVALID$TYPE,
3                   CALL SET$VALUE2(NEXT$AVAILABLE),
3                   CALL CODE$BYTE(CAL),
3                   CALL CODE$ADDRESS(L$AODR);
3                   CALL CODE$ADDRESS(L$LENGTH),
3                   CALL SET$BRANCH;
3              END,
2              ELSE DO;
3                   IF NUMERIC$TYPE THEN CTR=1,
3                   ELSE CTR=0,
3                   H$TYPE=L$TYPE,
3                   H$DEC=L$DEC,
3                   H$ADDR=L$ADOR;
3                   H$LENGTH=L$LENGTH,
3                   CALL LOAO$L$IO(SP),
3                   IF NUMERIC$TYPE THEN CTR=CTR+1,
3                   IF CTR=2 THEN     /* NUMERIC COMPARE */
3                   DO;
4                        CALL LOAD$REG(0, MP),
4                        CALL SET$VALUE2(NEXT$AVAILABLE-6),
4                        CALL LOAD$REG(1, SP);
4                        CALL CODE$BYTE(SUB),
4                        CALL CODE$BYTE(RGT + COND$TYPE),
4                        CALL SET$BRANCH;
4                   ENO,
3                   ELSE DO,
                           * ALPHA NUMERIC COMPARE *
```

```
                    IF (H$DEC<>0) OR (H$TYPE=COMP)
                        OR (L$DEC<>0) OR (L$TYPE=COMP)
                        OR (H$LENGTH<L$LENGTH) THEN CALL INVALID$TYPE;
                    CALL SET$VALUE2(NEXT$AVAILABLE);
                    CALL CODE$BYTE(SGT+COND$TYPE);
                    CALL CODE$ADDRESS(H$ADDR);
                    CALL CODE$ADDRESS(L$ADDR);
                    CALL CODE$ADDRESS(H$LENGTH);
                    CALL SET$BRANCH;
                END;
            END;
        END GEN$COMPARE;


        MOVE$TYPE: PROCEDURE BYTE;
            DECLARE
            HOLD$TYPE BYTE,
            ALPHA$NUM$MOVE         LIT '0',
            A$N$ED$MOVE            LIT '1',
            NUMERIC$MOVE           LIT  '2',
            N$ED$MOVE              LIT  '3';

            L$TYPE=AND$OUT$OCCURS(L$TYPE);
            IF((HOLD$TYPE :=AND$OUT$OCCURS(GET$TYPE))=GROUP) OR (L$TYPE=GROUP)
                THEN RETURN ALPHA$NUM$MOVE;
            IF HOLD$TYPE=ALPHA THEN
                IF A$AN$TYPE OR (L$TYPE=A$ED) OR (L$TYPE=A$N$ED)
                    THEN RETURN ALPHA$NUM$MOVE;
            IF HOLD$TYPE=ALPHA$NUM THEN
            DO;
                IF NOT$INTEGER THEN CALL INVALID$TYPE;
                RETURN ALPHA$NUM$MOVE;
            END;
            IF (HOLD$TYPE>=NUMERIC) AND (HOLD$TYPE<=COMP) THEN
            DO;
                IF (L$TYPE=ALPHA) OR (L$TYPE>COMP) THEN CALL INVALID$TYPE;
                RETURN NUMERIC$MOVE;
            END;
            IF HOLD$TYPE=A$N$ED THEN
            DO;
                IF NOT$INTEGER THEN CALL INVALID$TYPE;
                RETURN A$N$ED$MOVE;
            END;
            IF HOLD$TYPE=A$ED THEN
                IF A$AN$TYPE OR (L$TYPE>COMP) THEN RETURN A$N$ED$MOVE;
            IF HOLD$TYPE=NUM$ED THEN
                IF NUMERIC$TYPE OR (L$TYPE=ALPHA$NUM) THEN
                    RETURN N$ED$MOVE;
            CALL INVALID$TYPE;
            RETURN 0;
        END MOVE$TYPE;


        GEN$MOVE PROCEDURE;
            DECLARE
            LENGTH1 ADDRESS,
            ADDR1 ADDRESS,
            EXTRA ADDRESS;

            ADD$ADD$LEN  PROCEDURE;
                CALL CODE$ADDRESS(ADDR1);
                CALL CODE$ADDRESS(L$ADDR);
                CALL CODE$ADDRESS(L$LENGTH);
            END ADD$ADD$LEN;

            CODE$FOR$EDIT  PROCEDURE;
                CALL ADD$ADD$LEN;
                CALL CODE$ADDRESS(GET$FCB$ADDR);
                CALL CODE$ADDRESS(LENGTH1);
            END CODE$FOR$EDIT;

            CALL LOAD$L$ID(MPP1);
            CUR$SYM=VALUE(SP);
            IF (ADDR1:=VALUE2(SP))=0 THEN ADDR1=GET$ADDRESS;
            LENGTH1=GET$LENGTH;

            DO CASE MOVE$TYPE;

                  /* ALPHA NUMERIC MOVE */

                DO;
                    IF LENGTH1>L$LENGTH THEN EXTRA=LENGTH1-L$LENGTH;
                    ELSE DO;
                        EXTRA=0;
                        L$LENGTH=LENGTH1;
                    END;
                    CALL CODE$BYTE(MOV);
```

```
  4                   CALL ADD$ADD$LEN;
  4                   CALL CODE$ADDRESS(EXTRA);
  4              END;

                 /* ALPHA NUMERIC EDITED */

  3              DO;
  4                   CALL CODE$BYTE(MED);
  4                   CALL CODE$FOR$EDIT;
  4              END;

                 /* NUMERIC MOVE */

  3              DO;
  4                   CALL LOAD$REG(2,MPP1);
  4                   CALL STORE$REG(SP);
  4              END;

                 /* NUMERIC EDITED MOVE */

  3              DO;
  4                   CALL CODE$BYTE(MNE);
  4                   CALL CODE$FOR$EDIT;
  4                   CALL CODE$BYTE(L$DEC);
  4                   CALL CODE$BYTE(GET$DECIMAL);
  4              END;
  3         END;
  2     END GEN$MOVE;


  1     CODE$GEN: PROCEDURE(PRODUCTION);
  2         DECLARE PRODUCTION BYTE;
  2         IF PRINT$PROD THEN
  2         DO;
  3              CALL CRLF;
  3              CALL PRINTCHAR(POUND);
  3              CALL PRINT$NUMBER(PRODUCTION);
  3         END;

  2         DO CASE PRODUCTION;

        /*  P R O D U C T I O N S */

            /* CASE 0 NOT USED  */
  3                  ;

       /*       1   <P-DIV>   = PROCEDURE DIVISION <USING>   <PROC-BODY>       */

  3         DO;
  4              COMPILING = FALSE;
  4              IF SECTION$FLAG THEN CALL LOAD$SEC$LABEL;
  4         END;

       /*       2   <USING>    = USING <ID-STRING>                             */

  3         CALL NOT$IMPLIMENTED;     /* INTER PROG COMM */

       /*       3                \! <EMPTY>                                     */

  3          ;       /* NO ACTION REQUIRED */

       /*       4   <ID-STRING>    = <ID>                                      */

  3         ID$STACK(ID$PTR :=0)=VALUE(SP);
       /*       5                   \! <ID-STRING> <ID>                         */

  3         DO;
  4              IF(ID$PTR :=IDPTR+1)=20 THEN
  4              DO;
  5                   CALL PRINT$ERROR('ID');
  5                   ID$PTR=19;
  5              END;
  4              ID$STACK(ID$PTR)=VALUE(SP);
  4         END;

       /*       6   <PROC-BODY>    = <PARAGRAPH>                               */

  3          ;       /* NO ACTION REQUIRED */

       /*       7                \! <PROC-BODY> <PARAGRAPH>                     */

  3          ;       /* NO ACTION REQUIRED */

       /*       8   <PARAGRAPH>    = <ID>    <SENTENCE-LIST>                   */

  3         DO;
```

```
                IF SECTION$FLAG=0 THEN SECTION$FLAG=2;
                CALL LOAD$LABEL;
           END;

      /*        9                   \! <ID> SECTION                               */

           DO;
                IF SECTION$FLAG<>1 THEN
                DO;
                     IF SECTION$FLAG=2 THEN CALL PRINT$ERROR('PF');
                     SECTION$FLAG=1;
                     HOLD$SECTION=VALUE(MP);
                     HOLD$SEC$ADDR=VALUE2(MP);
                END;
                ELSE CALL LOAD$SEC$LABEL;
           END;

      /*       10   <SENTENCE-LIST>  ::= <SENTENCE>                              */

           ;    /* NO ACTION REQUIRED */

      /*       11                      \! <SENTENCE-LIST> <SENTENCE>              */

           ;    /* NO ACTION REQUIRED */

      /*       12   <SENTENCE>    = <IMPERATIVE>                                  */

           ;    /* NO ACTION REQUIRED */

      /*       13                   \! <CONDITIONAL>                              */

           ;    /* NO ACTION REQUIRED */

      /*       14                   \! ENTER <ID> <OPT-ID>                        */

           CALL NOT$IMPLIMENTED;     /* LANGUAGE CHANGE */

      /*       15   <IMPERATIVE>    = ACCEPT <SUBID>                              */

           DO;
               CALL LOAD$L$ID(SP);
               CALL ONE$ADDR$OPP(ACC,L$ADDR);
               CALL CODE$BYTE(L$LENGTH);
           END;

      /*       16                     \! <ARITHMETIC>                             */

           ;    /* NO ACTION REQUIRED */

      /*       17                     \! CALL <LIT> <USING>                       */

          CALL NOT$IMPLIMENTED;     /* INTER PROG COMM */

      /*       18                     \! CLOSE <ID>                               */

           CALL ONE$ADDR$OPP(CLS,GET$FCB$ADDR);

      /*       19                     \! <FILE-ACT>                               */

           ;    /* NO ACTION REQUIRED */

      /*       20                     \! DISPLAY <LIT/ID> <OPT-LIT/ID>            */

           DO;
                CALL CODE$FOR$DISPLAY(MPP1);
                IF VALUE(SP)<>0 THEN CALL CODE$FOR$DISPLAY(SP);
           END;

      /*       21                     \! EXIT <PROGRAM-ID>                        */

           ;    /* NO ACTION REQUIRED */

      /*       22                     \! GO <ID>                                  */

           CALL ONE$ADDR$OPP(BRN,LABEL$ADDR(VALUE(SP),1));

      /*       23                     \! GO <ID-STRING> DEPENDING <ID>            */

           DO;
                CALL CODE$BYTE(GDP);
                CALL CODE$BYTE(ID$PTR);
                CUR$SYM=VALUE(SP);
                CALL CODE$BYTE(GET$LENGTH);
                CALL CODE$ADDRESS(GET$ADDRESS);
                DO CTR=0 TO ID$PTR;
                     CALL CODE$ADDRESS(LABEL$ADDR$OFFSET(ID$STACK(ID$PTR),1,0));
                END;
```

```
879   4          END;

              /*      24                   \! MOVE <LIT/ID> TO <SUBID>                    */

880   3          CALL GEN$MOVE;

              /*      25                   \! OPEN <TYPE-ACTION> <ID>                     */

881   3          CALL ONE$ADDR$OPP(OPN + VALUE(MPP1), GET$FCB$ADDR);

              /*      26                   \! PERFORM <ID> <THRU> <FINISH>                */

882   3          DO;
883   4               DECLARE (ADDR2, ADDR3) ADDRESS;
884   4               IF VALUE(SP-1)=0 THEN ADDR2=LABEL$ADDR$OFFSET(VALUE(MPP1), 0, 3);
886   4               ELSE ADDR2=LABEL$ADDR$OFFSET(VALUE(SP-1), 0, 3);
887   4               IF (ADDR3 =VALUE2(SP))=0 THEN ADDR3=NEXT$AVAILABLE + 7;
889   4               ELSE CALL BACKSTUFF(VALUE(SP), NEXT$AVAILABLE + 7);
890   4               CALL ONE$ADDR$OPP(PER, LABEL$ADDR(VALUE(MPP1), 1));
891   4               CALL CODE$ADDRESS(ADDR2);
892   4               CALL CODE$ADDRESS(ADDR3);
893   4          END;

              /*      27                   \! <READ-ID>                                   */

894   3          CALL NOT$IMPLIMENTED;     /* GRAMMAR ERROR */

              /*      28                   \! STOP <TERMINATE>                            */

895   3          DO;
896   4               IF VALUE(SP)=0 THEN CALL CODE$BYTE(STP);
898   4           ELSE DO;
899   5              CALL ONE$ADDR$OPP(STD, VALUE2(SP));
900   5              CALL CODE$BYTE(CON$LENGTH);
901   5           END;
902   4          END;

              /*      29   <CONDITIONAL>    = <ARITHMETIC> <SIZE-ERROR>                  */
              /*      29                      <IMPERATIVE>                               */

903   3          CALL BACK$COND;

              /*      30                    \! <FILE-ACT> <INVALID> <IMPERATIVE>          */

904   3          CALL BACK$COND;

              /*      31                    \! <IF-NONTERMINAL> <CONDITION> <ACTION> ELSE        */
              /*      31                      <IMPERATIVE>                               */

905   3          DO;
906   4               CALL BACKSTUFF(VALUE(MPP1), VALUE2(SP-2));
907   4               CALL BACKSTUFF(VALUE(SP-2), NEXT$AVAILABLE);
908   4          END;

              /*      32                    \! <READ-ID> <SPECIAL> <IMPERATIVE>           */

909   3          CALL BACK$COND;

              /*      33   <ARITHMETIC>    = ADD <L/ID> <OPT-L/ID> TO <SUBID>           */
              /*      33                     <ROUND>                                     */

910   3          CALL ADD$SUB(0);

              /*      34                   \! DIVIDE <L/ID> INTO <SUBID> <ROUND>          */

911   3          CALL MULT$DIV(1);

              /*      35                   \! MULTIPLY <L/ID> BY <SUBID> <ROUND>          */

912   3          CALL MULT$DIV(0);

              /*      36                   \! SUBTRACT <L/ID> <OPT-L/ID> FROM             */
              /*      36                     <SUBID> <ROUND>                             */

913   3          CALL ADD$SUB(1);

              /*      37   <FILE-ACT>    = DELETE <ID>                                   */

914   3          CALL DEL$RWT(0);

              /*      38                 \! REWRITE <ID>                                  */

915   3          CALL DEL$RWT(1);

              /*      39                 \! WRITE <ID> <SPECIAL-ACT>                      */

916   3          CALL READ$WRITE(1);
```

```
                /*      40   <CONDITION>    = <LIT/ID> <NOT> <COND-TYPE>                      */

3                 DO;
4                   IF IF$FLAG THEN
4                     DO;
5                       IF$FLAG=NOT IF$FLAG;         /* RESET IF$FLAG */
5                       CALL CODE$BYTE(NEG);
5                     END;
4                   CALL GEN$COMPARE;
4                 END;

                /*      41   <COND-TYPE>    = NUMERIC                                         */

3                   COND$TYPE=3;

                /*      42                  \! ALPHABETIC                                     */

3                   COND$TYPE=4;

                /*      43                  \! <COMPARED> <LIT/ID>                            */

3                   CALL KEEP$VALUES;

                /*      44   <NOT> :  = NOT                                                   */

3                   IF NOT IF$FLAG THEN
3                   CALL CODE$BYTE(NEG);
3                   ELSE IF$FLAG=NOT IF$FLAG;         /* RESET IF$FLAG */

                /*      45             \! <EMPTY>                                             */

3                   ;    /* NO ACTION REQUIRED */

                /*      46   <COMPARE>     = GREATER                                          */

3                   COND$TYPE=0;

                /*      47                 \! LESS                                            */

3                   COND$TYPE=1;

                /*      48                 \! EQUAL                                           */

3                   COND$TYPE=2;

                /*      49   <ROUND>  = ROUNDED                                               */

3                   CALL SET$VALUE(1);

                /*      50                 \! <EMPTY>                                         */

3                   ;    /* NO ACTION REQUIRED */

                /*      51   <TERMINATE>    = <LITERAL>                                       */

3                   ;    /* NO ACTION REQUIRED */

                /*      52                  \!.RUN                                            */

3                   ;    /* NO ACTION REQUIRED - VALUE(SP) ALREADY ZERO */

                /*      53   <SPECIAL>     = <INVALID>                                        */

3                   ;    /* NO ACTION REQUIRED */

                /*      54                 \! END                                             */

3                   DO;
4                        CALL SET$VALUE(2);
4                        CALL CODE$BYTE(EOR);
4                        CALL SET$BRANCH;
4                   END;

                /*      55   <OPT-ID>     = <SUBID>                                           */

3                   ;    /* VALUE AND VALUE2 ALREADY SET */

                /*      56                \!                                                  */

3                   ;    /* VALUE ALREADY ZERO */

                /*      57   <ACTION>     = <IMPERATIVE>                                      */

3                   CALL UNRESOLVED$BRANCH;

                /*      58                \! NEXT SENTENCE                                    */
```

```
3           CALL UNRESOLVED$BRANCH;

        /*      59    <THRU>     = THRU <ID>                                          */

3           CALL KEEP$VALUES;

        /*      60                \!                                                   */

3           ;    /* NO ACTION REQUIRED */

        /*      61    <FINISH>     = <L/ID> TIMES                                      */

3           DO;
4               CALL LOAD$L$ID(MP);
4               CALL ONE$ADDR$OPP(LDI,L$ADDR);
4               CALL CODE$BYTE(L$LENGTH);
4               CALL SET$VALUE2(NEXT$AVAILABLE);
4               CALL ONE$ADDR$OPP(DEC,0);
4               CALL SET$VALUE(NEXT$AVAILABLE);
4           CALL CODE$ADDRESS(0);      END;

        /*      62                  \! UNTIL <CONDITION>                               */

3           CALL KEEP$VALUES;

        /*      63                  \!                                                 */

3           ;    /* NO ACTION REQUIRED */

        /*      64    <INVALID>     = INVALID                                          */

3           DO;
4               CALL SET$VALUE(1);
4               CALL CODE$BYTE(INV);
4               CALL SET$BRANCH;
4           END;

        /*      65    <SIZE-ERROR>  = SIZE ERROR                                       */

3           DO;
4               CALL CODE$BYTE(SER);
4               CALL UNRESOLVED$BRANCH;
4           END;

        /*      66    <SPECIAL-ACT>   = <WHEN> ADVANCING <HOW-MANY>                    */

3           CALL NOT$IMPLIMENTED;      /* CARRAGE CONTROL */

        /*      67                   \!                                                */

3           ;    /* NO ACTION REQUIRED */

        /*      68    <WHEN>     = BEFORE                                              */

3           CALL NOT$IMPLIMENTED;      /* CARRAGE CONTROL */

        /*      69                \! AFTER                                             */

3           CALL NOT$IMPLIMENTED;      /* CARRAGE CONTROL */

        /*      70    <HOW-MANY>    = <INTEGER>                                        */

3           CALL NOT$IMPLIMENTED;      /* CARRAGE CONTROL */

        /*      71                  \! PAGE                                            */

3           CALL NOT$IMPLIMENTED;      /* CARRAGE CONTROL */

        /*      72    <TYPE-ACTION>    = INPUT                                         */

3           ;    /* NO ACTION REQUIRED - VALUE(SP) ALREADY ZERO */

        /*      73                   \! OUTPUT                                         */

3           CALL  SET$VALUE(1);

        /*      74                   \! I-O                                            */

3           CALL SET$VALUE(2);

        /*      75    <SUBID>     = <SUBSCRIPT>                                        */

3           ;    /* VALUE AND VALUE2 ALREADY SET */

        /*      76                \! <ID>                                              */
```

```
 981   3          ;     /* NO ACTION REQUIRED */

               /*     77   <INTEGER>    = <INPUT>                                  */

 982   3        CALL SET$VALUE(CONVERT$INTEGER);

               /*     78   <ID>     = <INPUT>                                      */

 983   3        DO;
 984   4             CALL SET$VALUE(MATCH);
 985   4             IF GET$TYPE=UNRESOLVED THEN CALL SET$VALUE2(NEXT$AVAILABLE);
 987   4        END;

               /*     79   <L/ID>     = <INPUT>                                    */

 988   3        DO;
 989   4             IF NUMERIC$LIT THEN
 990   4             DO;
 991   5                  CALL SET$VALUE(NUMERIC$LITERAL);
 992   5                  CALL SET$VALUE2(STORE$CONSTANT);
 993   5             END;
 994   4             ELSE CALL SET$VALUE(MATCH);
 995   4        END;

               /*     80              \! <SUBSCRIPT>                                */

 996   3          ;     /* NO ACTION REQUIRED */

               /*     81              \! ZERO                                        */

 997   3        CALL SET$VALUE(LIT$ZERO);

               /*     82   <SUBSCRIPT>    = <ID> ( <INPUT> )                         */

 998   3        CALL CHECK$SUBSCRIPT;

               /*     83   <OPT-L/ID>     = <L/ID>                                   */

 999   3          ;     /* NO ACTION REQUIRED */

               /*     84                  \! <EMPTY>                                 */

1000   3          ;     /* VALUE ALREADY SET */

               /*     85   <NN-LIT>    := <LIT>                                      */

1001   3        DO;
1002   4             CALL SET$VALUE(NON$NUMERIC$LIT);
1003   4             CALL SET$VALUE2(STORE$CONSTANT);
1004   4        END;

               /*     86                \! SPACE                                     */

1005   3        CALL SET$VALUE(LIT$SPACE);

               /*     87                \! QUOTE                                     */

1006   3        CALL SET$VALUE(LIT$QUOTE);

               /*     88   <LITERAL>    = <NN-LIT>                                   */

1007   3          ;     /* NO ACTION REQUIRED */

               /*     89                 \! <INPUT>                                  */

1008   3        DO;
1009   4             IF NOT NUMERIC$LIT THEN CALL INVALID$TYPE;
1011   4             CALL SET$VALUE(NUMERIC$LITERAL);
1012   4             CALL SET$VALUE2(STORE$CONSTANT);
1013   4        END;

               /*     90                 \! ZERO                                     */

1014   3        CALL SET$VALUE(LIT$ZERO);

               /*     91   <LIT/ID>     = <L/ID>                                     */

1015   3          ;     /* NO ACTION REQUIRED */

               /*     92                \! <NN-LIT>                                  */

1016   3          ;     /* NO ACTION REQUIRED */

               /*     93   <OPT-LIT/ID>     = <LIT/ID>                               */

1017   3          ;     /* NO ACTION REQUIRED */
```

```
            /*      94                    \! <EMPTY>                          */
    3           ;     /* NO ACTION REQUIRED */

            /*      95   <PROGRAM-ID>    = <ID>                               */
    3           CALL NOT$IMPLIMENTED;     /* INTER PROG COMM */

            /*      96                    \!                                  */
    3           ;     /* NO ACTION REQUIRED */

            /*      97   <READ-ID>     = READ <ID>                            */
    3           CALL READ$WRITE(0);

            /*        98    <IF-NONTERMINAL>    = IF               */
    3           IF$FLAG = TRUE;            /* SET IF$FLAG */

    3           END;    /* END OF CASE STATEMENT */
    2       END CODE$GEN;

    1       GETIN1: PROCEDURE BYTE;
    2           RETURN INDEX1(STATE);
    2       END GETIN1;

    1       GETIN2: PROCEDURE BYTE;
    2           RETURN INDEX2(STATE);
    2       END GETIN2;

    1       INCSP: PROCEDURE;
    2           VALUE(SP :=SP + 1)=0;      /* CLEAR THE STACK WHILE INCREMENTING */
    2           VALUE2(SP)=0;
    2           IF SP >= PSTACKSIZE THEN CALL FATAL$ERROR('SO');
    2       END INCSP;

    1       LOOKAHEAD: PROCEDURE;
    2           IF NOLOOK THEN
    2           DO;
    3                CALL SCANNER;
    3                NOLOOK=FALSE;
    3                IF PRINT$TOKEN THEN
    3                DO;
    4                     CALL CRLF;
    4                     CALL PRINT$NUMBER(TOKEN);
    4                     CALL PRINT$CHAR(' ');
    4                     CALL PRINT$ACCUM;
    4                END;
    3           END;
    2       END LOOKAHEAD;

    1       NO$CONFLICT: PROCEDURE (CSTATE) BYTE;
    2           DECLARE (CSTATE, I, J, K) BYTE;
    2           J=INDEX1(CSTATE);
    2           K=J + INDEX2(CSTATE) - 1;
    2           DO I=J TO K;
    3                IF READ1(I)=TOKEN THEN RETURN TRUE;
    3           END;
    2       RETURN FALSE;
    2       END NO$CONFLICT;

    1       RECOVER: PROCEDURE BYTE;
    2           DECLARE TSP BYTE, RSTATE BYTE;
    2           DO FOREVER;
    3                TSP=SP;
    3                DO WHILE TSP <> 255;
    4                     IF NO$CONFLICT(RSTATE :=STATESTACK(TSP)) THEN
    4                     DO;   /* STATE WILL READ TOKEN */
    5                          IF SP<>TSP THEN SP = TSP - 1;
    5                          RETURN RSTATE;
    5                     END;
    4                     TSP = TSP - 1;
    4                END;
    3                CALL SCANNER;   /* TRY ANOTHER TOKEN */
    3           END;
    2       END RECOVER;

                /*  *   *   *   *    PROGRAM EXECUTION STARTS HERE *   *   */


                /* INITIALIZATION */

    1       TOKEN=53;       /* PRIME THE SCANNER WITH -PROCEDURE- */
    1       CALL MOVE(PASS1$TOP-PASS1$LEN, OUTPUT$FCB, PASS1$LEN);
                /* THIS SETS
                     OUTPUT FILE CONTROL BLOCK
                     TOGGLES
```

```
                    READ POINTER
                    NEXT SYMBOL TABLE POINTER
                 */
  1       OUTPUT$END=(OUTPUT$PTR = OUTPUT$BUFF-1)+128;

              /* * * * * * * PARSER * * * * * */

  1       DO WHILE COMPILING;
  2           IF STATE <= MAXRNO THEN        /* READ STATE */
  2           DO;
  3               CALL INCSP;
  3               STATESTACK(SP) = STATE;   /* SAVE CURRENT STATE */
  3               CALL LOOKAHEAD;
  3               I=GETIN1;
  3               J = I + GETIN2 - 1;
  3               DO I=I TO J;
  4                   IF READ1(I) = TOKEN THEN
  4                   DO;
                      /* COPY THE ACCUMULATOR IF IT IS AN INPUT
                      STRING.  IF IT IS A RESERVED WORD IT DOES
                      NOT NEED TO BE COPIED  */
  5                       IF (TOKEN=INPUT$STR) OR (TOKEN=LITERAL) THEN
  5                           DO K=0 TO ACCUM(0);
  6                               VARC(K)=ACCUM(K);
  6                           END;
  5                       STATE=READ2(I);
  5                       NOLOOK=TRUE;
  5                       I=J;
  5                   END;
                      ELSE
  4                   IF I=J THEN
  4                   DO;
  5                       CALL PRINT$ERROR('NP');
  5                       CALL PRINT( ERROR$NEAR$$);
  5                       CALL PRINT$ACCUM;
  5                       IF (STATE:=RECOVER)=0 THEN COMPILING=FALSE;
  5                   END;
                  END;
  3           END;    /* END OF READ STATE */
              ELSE
  2           IF STATE>MAXPNO THEN      /* APPLY PRODUCTION STATE */
  2           DO;
  3               MP=SP - GETIN2;
  3               MPP1=MP + 1;
  3               CALL CODE$GEN(STATE - MAXPNO);
  3               SP=MP;
  3               I=GETIN1;
  3               J=STATESTACK(SP);
  3               DO WHILE (K:=APPLY1(I)) <> 0 AND J<>K;
  4                   I=I + 1;
  4               END;
  3               IF (K:=APPLY2(I))=0 THEN COMPILING=FALSE;
  3               STATE=K;
  3           END;
              ELSE
  2           IF STATE<=MAXLNO THEN     /*LOOKAHEAD STATE*/
  2           DO;
  3           I=GETIN1;
  3               CALL LOOKAHEAD;
  3               DO WHILE (K :=LOOK1(I))<>0 AND TOKEN <>K;
  4                   I=I+1;
  4               END;
  3           STATE=LOOK2(I);
  3           END;
              ELSE
  2           DO;         /*PUSH STATES*/
  3               CALL INCSP;
  3               STATESTACK(SP)=GETIN2;
  3               STATE=GETIN1;
  3           END;
  2       END;   /* OF WHILE COMPILING */
  1       CALL BYTE$OUT(TER);
  1       DO WHILE OUTPUT$PTR<> OUTPUT$BUFF;
  2           CALL BYTE$OUT(TER);
  2       END;
  1       CALL CLOSE;
  1       CALL CRLF;
  1       CALL PRINT( END$OF$PART$2);
  1       CALL BOOT;
  1       END;


MODULE INFORMATION

     CODE AREA SIZE     = 1030H   8251D
```

```
ISIS-II PL/M-80 V3.1 COMPILATION OF MODULE DECODE
OBJECT MODULE PLACED IN  F1 DECODE.OBJ
COMPILER INVOKED BY   PLM80  F1 DECODE.PLM


              $PAGELENGTH(90)
   1          DECODE.  DO,

                  /* THIS PROGRAM TAKES THE CODE OUTPUT FROM THE COBOL COMPILER
                  AND CONVERTS IT INTO A READABLE OUTPUT TO FACILITATE DEBUGGING */

              /* * * 100H         LOAD POINT */

   2   1      DECLARE

              LIT           LITERALLY       'LITERALLY',
              BOOT          LIT             '0',
              BDOS          LIT             '5',
              FCB           ADDRESS         INITIAL (5CH),
              FCB$BYTE      BASED      FCB (1)  BYTE,
              I             BYTE,
              ADDR          ADDRESS         INITIAL (100H),
              CHAR          BASED      ADDR BYTE,
              C$ADDR        BASED ADDR            ADDRESS,
              BUFF$END      LIT             '0FFH',
              FILE$TYPE (*) BYTE      DATA ('C','I','N'),

   3   1      MON1. PROCEDURE (F,A),
   4   2          DECLARE F BYTE, A ADDRESS,
   5   2          L   GO TO L,   /*    PATCH TO JMP 5   */
   6   2      END MON1,

   7   1      MON2. PROCEDURE (F,A) BYTE,
   8   2          DECLARE F BYTE, A ADDRESS,
   9   2        L GO TO L,     /* * *    PATCH TO    " JMP 5 "   * * */
  10   2          RETURN 0;
  11   2      END MON2,

  12   1      PRINT$CHAR. PROCEDURE(CHAR);
  13   2          DECLARE CHAR BYTE;
  14   2          CALL MON1(2,CHAR),
  15   2 .    END PRINT$CHAR;

  16   1      CRLF. PROCEDURE,
  17   2          CALL PRINT$CHAR(13),
  18   2          CALL PRINT$CHAR(10);
  19   2      END CRLF,

  20   1      P: PROCEDURE(ADD1),
  21   2          DECLARE ADD1 ADDRESS, C BASED ADD1 (1) BYTE,
  22   2          CALL CRLF,
  23   2          DO I=0 TO 2,
  24   3              CALL PRINT$CHAR(C(I)),
  25   3          END;
  26   2          CALL PRINT$CHAR(' '),
  27   2      END P;

  28   1      GET$CHAR. PROCEDURE BYTE,
  29   2          IF (ADDR =ADDR + 1)>BUFF$END THEN
  30   2          DO;
  31   3              IF MON2(20,FCB)<>0 THEN
  32   3              DO,
  33   4                  CALL P(.('END')),
  34   4                  CALL TIME(10);
  35   4                L.  GO TO L,     /*  PATCH TO  "JMP 0000"   * * * */
  36   4              END,
  37   3              ADDR=80H,
  38   3          END;
  39   2        . RETURN CHAR,
  40   2      END GET$CHAR,

  41   1      D$CHAR  PROCEDURE (OUTPUT$BYTE),
  42   2          DECLARE OUTPUT$BYTE BYTE,
  43   2          IF OUTPUT$BYTE<10 THEN CALL PRINT$CHAR(OUTPUT$BYTE + 30H),
  45   2          ELSE CALL PRINT$CHAR(OUTPUT$BYTE + 37H),
  46   2      END D$CHAR,

  47   1      D  PROCEDURE (COUNT),
  48   2          DECLARE(COUNT,J) ADDRESS,
  49   2          DO J=1 TO COUNT
```

```
 3              CALL D$CHAR(SHR(GET$CHAR,4));
 3              CALL D$CHAR(CHAR AND 0FH);
 3              CALL PRINT$CHAR(' ');
 3          END;
 2      END D;

 1      PRINT$REST: PROCEDURE;
 2          DECLARE
            F2   LIT  '9',
            F3   LIT  '9',
            F4   LIT  '21',
            F5   LIT  '24',
            F6   LIT  '32',
            F7   LIT  '39',
            F9   LIT  '49',
            F10  LIT  '54',
            F11  LIT  '60',
            F12  LIT  '61',
            GDP  LIT  '62',
            INT  LIT  '63',
            BST  LIT  '64',
            TER  LIT  '65',
            SCD  LIT  '66';

 2          IF CHAR < F2 THEN RETURN;
 2          IF CHAR < F3 THEN DO; CALL D(1); RETURN; END;
 2          IF CHAR < F4 THEN DO; CALL D(2); RETURN; END;
 2          IF CHAR < F5 THEN DO; CALL D(3); RETURN; END;
 2          IF CHAR < F6 THEN DO; CALL D(4); RETURN; END;
 2          IF CHAR < F7 THEN DO; CALL D(5); RETURN; END;
 2          IF CHAR < F9 THEN DO; CALL D(6); RETURN; END;
 2          IF CHAR < F10 THEN DO; CALL D(8); RETURN; END;
 2          IF CHAR < F11 THEN DO; CALL D(9); RETURN; END;
 2          IF CHAR < F12 THEN DO; CALL D(10); RETURN; END;
 2          IF CHAR < GDP THEN DO; CALL D(12); RETURN; END;
 2          IF CHAR=GDP THEN DO;
 3       CALL D(1); CALL D(SHL(CHAR,1)+5); RETURN; END;
 2          IF CHAR=INT THEN DO; CALL D(3); CALL D(C$ADDR + 1); RETURN; END;
 2          IF CHAR=BST THEN DO; CALL D(4); RETURN; END;
 2          IF CHAR=TER THEN DO; CALL P(.('END'));
 3         L: GO TO L;   /* PATCH TO  "JMP  0"  * * */ END;
 2          IF CHAR=SCD THEN DO; CALL D(2); RETURN; END;
 2          IF CHAR <> 0FFH THEN CALL P(.('XXX'));
 2      END PRINT$REST;

.
            /* PROGRAM EXECUTION STARTS HERE */

 1      FCB$BYTE(32), FCB$BYTE(0) = 0;
 1      DO I=0 TO 2;
 2          FCB$BYTE(I+9)=FILE$TYPE(I);
 2      END;

 1      IF MON2(15,FCB)=255 THEN DO; CALL P(.('ZZZ'));
 2                                   L: GO TO L; END;
                                       /* * * * PATCH TO  "JMP BOOT" * * * */

 1 .    DO WHILE 1;
 2          IF GET$CHAR <= 66 THEN DO CASE CHAR;
 3          ;     /* CASE 0 NOT USED */
 3                CALL P(.('ADD'));
 3                CALL P(.('SUB'));
 3                CALL P(.('MUL'));
 3                CALL P(.('DIV'));
 3                CALL P(.('NEG'));
 3                CALL P(.('STP'));
 3                CALL P(.('STI'));
 3                CALL P(.('RND'));
 3                CALL P(.('RET'));
 3                CALL P(.('CLS'));
 3                CALL P(.('SER'));
 3                CALL P(.('BRN'));
 3                CALL P(.('OPN'));
 3                CALL P(.('OP1'));
 3                CALL P(.('OP2'));
 3                CALL P(.('RGT'));
 3                CALL P(.('RLT'));
 3                CALL P(.('REQ'));
 3                CALL P(.('INV'));
 3                CALL P(.('EOR'));
 3                CALL P(.('ACC'));
 3                CALL P(.('DIS'));
 3                CALL P(.('STD'));
 3                CALL P(.('LDI'));
 3                CALL P(.('DEC'));
 3                CALL P(.('STO'));
```

```
 3              CALL P(.('ST1'));
 3              CALL P(.('ST2'));
 3              CALL P(.('ST3'));
 3              CALL P(.('ST4'));
 3              CALL P(.('ST5'));
 3              CALL P(.('LD0'));
 3              CALL P(.('LD1'));
 3              CALL P(.('LD2'));
 3              CALL P(.('LD3'));
 3              CALL P(.('LD4'));
 3              CALL P(.('LD4'));
 3              CALL P(.('LD6'));
 3              CALL P(.('PER'));
 3              CALL P(.('CNU'));
 3              CALL P(.('CNS'));
 3              CALL P(.('CAL'));
 3              CALL P(.('RWS'));
 3              CALL P(.('DLS'));
 3              CALL P(.('RDF'));
 3              CALL P(.('WTF'));
 3              CALL P(.('RVL'));
 3              CALL P(.('WVL'));
 3              CALL P(.('SCR'));
 3              CALL P(.('SGT'));
 3              CALL P(.('SLT'));
 3              CALL P(.('SEQ'));
 3              CALL P(.('MOV'));
 3              CALL P(.('RRS'));
 3              CALL P(.('WRS'));
 3              CALL P(.('RRR'));
 3              CALL P(.('WRR'));
 3              CALL P(.('RWR'));
 3              CALL P(.('DLR'));
 3              CALL P(.('MED'));
 3              CALL P(.('MNE'));
 3              CALL P(.('GDP'));
 3              CALL P(.('INT'));
 3              CALL P(.('BST'));
 3              CALL P(.('TER'));
 3              CALL P(.('SCD'));
 3          END; /* OF CASE STATEMENT */
 2          CALL PRINT$REST;
 2      END; /* END OF DO WHILE */
 1      END;


MODULE INFORMATION

     CODE AREA SIZE     = 0671H   1649D
     VARIABLE AREA SIZE = 0013H     19D
     MAXIMUM STACK SIZE = 000EH     14D
     213 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-80 COMPILATION
```

LIST OF REFERENCES

1. Craig, Allen S. MICRO-COBOL An Implementation of Navy Standard Hypo-Cobol for a Micro-processor based Computer System.

2. Aho, A. V. and S. C. Johnson, LR Parsing, Computing Surveys, Vol. 6 No. 2, June 1974.

3. Bauer, F. L. and J. Eickel, editors, Compiler Construction — An Advanced Course, Lecture notes is Computer Science, Springer-Verlag, New York 1976.

4. Digital Research, An Introduction to CP/M Features and Facilities, 1976.

5. Digital Research, CP/M Interface Guide, 1976.

6. Eubanks, Gordon E. Jr. A Microprocessor Implementation of Extended Basic, Masters Thesis, Naval Postgraduate School, December 1976.

7. Intel Corporation, 8008 and 8080 PL/M Programming Manual, 1975.

8. Intel Corporation, 8080 Simulator Software Package, 1974.

9. Knuth, Donald E. On the Translation of Languages from Left to Right, Information and Control Vol. 8, No. 6, 1965.

10. Software Development Division, ADPE Selection Office, Department of the Navy, HYPO-COBOL, April 1975.

11. University of Toronto, Computer Systems Research Group Technical Report CSRG-2, "An Efficient LALR Parser Generator," by W. R. Lalonge, April 1971.

12. Digital Research, Symbolic Instruction Debugger User's Guide, 1978.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	3
4.	Assoc. Professor G. A. Kildall, Code 52Kd Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	Lt. M. S. Moranville, Code 52Ms Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6.	ADPE Selection Office Department of the Navy Washington, D. C. 20376	1
7.	P.R. Mylet 8005 Kidd St. Alexandria, Va. 22309	1

Thesis
M998 Mylet 177913
c.1 MICRO-COBOL: a subset of Navy standard HYPO-COBOL for micro-computers.